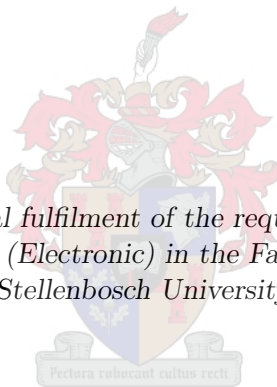


Optimised Path Planning and Path Tracking for Autonomous Vehicles with Constrained Kinematics in ROS

by

Bongani Bright Maseko

*Thesis presented in partial fulfilment of the requirements for the degree of
Master of Engineering (Electronic) in the Faculty of Engineering at
Stellenbosch University*



Supervisors:

Dr. C. E. van Daalen Mr. J. Treunicht
Autonomous Navigation and Mapping Research Group,
Electronic Systems Laboratory,
Department of Electrical and Electronic Engineering,
Stellenbosch University,
Private Bag X1, Matieland 7602, South Africa.

March 2020

Plagiarism Declaration

1. I have read and understand the Stellenbosch University Policy on Plagiarism and the definitions of plagiarism and self-plagiarism contained in the Policy [Plagiarism: The use of the ideas or material of others without acknowledgement, or the re-use of one's own previously evaluated or published material without acknowledgement or indication thereof (self-plagiarism or text-recycling)].
2. I agree that plagiarism is a punishable offence because it constitutes theft.
3. I also understand that direct translations are plagiarism.
4. Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.
5. By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Name: Bongani Bright Maseko

Date: March 2020

Copyright © 2020 Stellenbosch University
All rights reserved

Abstract

A significant growth of interest in the pursuit of autonomous vehicles from various stakeholders has been witnessed recently. This indicates that future transportation systems will be autonomous.

By the nature of the transportation problem, a transportation system is favourable if it is fast, safe and reliable. Therefore, if autonomous vehicles are to offer a genuinely superior alternative to present-day transportation systems, it is imperative that they do better than present-day transport in all these criteria.

This thesis focusses on two submodules of a typical autonomous navigation system that play a critical role in the fulfilment of these criteria. These submodules are path planning and path tracking. Path planning generates safe and optimal paths that result in reaching destinations safely and in minimal time. A path planner should also generate a plan quickly, otherwise delays are incurred. Path tracking is concerned with the accurate following of planned paths so that no collisions result. For a path tracker to accurately follow a planned path, it is necessary that the path be feasible for the target vehicle, otherwise both the path-planning and path-tracking efforts are in vain, as the vehicle will certainly deviate from the path and run the risk of collision.

Path-planning algorithms exist that plan paths *quickly and efficiently*. Such path planners have, however, been proven to be *almost-surely suboptimal*. At the other end of the spectrum, path-planning algorithms exist that are *guaranteed to find optimal paths*. However, their optimality guarantee hinges on the number of planning iterations *approaching infinity* – in technical terms they are said to be *asymptotically optimal*, with the practical implication being that they may run for unbearably long periods.

This thesis investigates the application of path optimisation to accelerate the rate of convergence of path-planning algorithms towards the optimal solution. The thesis first selects and develops suitable path-planning and path-optimisation algorithms to be used for the investigation. To ensure that the paths generated can be accurately executed by targeted vehicles, the developed path planners and path optimisers are adapted to incorporate motion constraints. The path optimisers are then incorporated into the various path planners with the aim of accelerating the rate of convergence and the effectiveness of each path optimiser in accelerating the convergence of each path planner is analysed. Of interest in this investigation is to ascertain if the application of path optimisation to accelerate the rate of convergence of the path planners helps a quick and efficient, though almost-surely suboptimal, path planner attain comparable or better performance than that of an asymptotically optimal path planner. Results obtained from the experiments indicate the affirmative.

Without demonstrating that the planned paths are indeed executable, the practical value of the developed optimised path-planning algorithms would be unclear. A path tracker has therefore been developed that accurately tracks planned paths in the absence of disturbances, and is able to correct for deviations when disturbances are encountered.

Opsomming

'n Beduidende toename in belangstelling in die ontwikkeling van outonome voertuie deur verskeie rolspelers is onlangs waargeneem. Dit dui aan dat vervoerstelsels in die toekoms outonoom gaan wees.

Uit die aard van die vervoerprobleem is 'n vervoerstelsel goed as dit vinnig, veilig en betroubaar is. Dus, vir outonome voertuie om regtig 'n beter alternatief vir hedendaagse vervoerstelsels te bied is dit nodig dat hulle beter as die hedendaagse vervoerstelsels werk vir al hierdie kriteria.

Hierdie tesis fokus op twee submodules van 'n tipiese outonome navigasiestelsel wat 'n kritiese rol speel om aan hierdie kriteria te voldoen. Hierdie submodules is padbeplanning en padvolging. Padbeplanning produseer veilige en optimale paaie wat daartoe lei dat bestemmings veilig en in 'n minimale tyd bereik word. 'n Padbeplanner moet ook paaie vinnig bereken, anders word vertraging veroorsaak. Padvolging het te doen met die akkurate volging van beplande paaie sodat geen botsings plaasvind nie. Vir 'n padvolger om 'n beplande pad akkuraat te volg, is dit nodig dat die pad uitvoerbaar is vir die voertuig, anders is die moeite van beide die padbeplanner en padvolger nutteloos, aangesien die voertuig sekerlik sal afwyk van die pad en moontlik botsings sal ervaar.

Daar bestaan padbeplanning-algoritmes wat paaie *vinnig en effektief* beplan. Dit is egter bewys dat hierdie padbeplanners *amper-seker suboptimaal* is. Aan die ander kant van die spektrum bestaan daar padbeplanning-algoritmes wat *gewaarborg is om optimale paaie te vind*. Hul optimaliteit-waarborg hang egter af daarvan dat die hoeveelheid beplannings-iterasies *na oneindig streef* – in tegniese terme word daar gesê dat hulle *asimptoties optimaal* is, met die praktiese gevolg dat hulle onaanvaarbare lang periodes kan neem om uit te voer.

Hierdie tesis ondersoek die toepassing van pad-optimeerders om die tempo van konvergensie van padbeplanning-algoritmes na die optimale oplossing te verhaas. Die tesis kies en ontwikkel eers toepaslike padbeplanning- en pad-optimering-algoritmes om te gebruik in die ondersoek. Om te verseker dat die paaie wat geproduseer word akkuraat uitgevoer kan word deur toepaslike voertuie, word die ontwikkelde padbeplanners en pad-optimeerders aangepas om bewegingsbeperkings in ag te neem. Die pad-optimeerders word dan saamgevoeg met die onderskeie padbeplanners met die doel om die tempo van konvergensie te verhaas, en die effektiwiteit van elke pad-optimeerder om die konvergensie van elke padbeplanner te verhaas word geanaliseer. 'n Doelwit van hierdie ondersoek is om te bepaal of die toepassing van pad-optimering om die tempo van konvergensie van die padbeplanners te verhaas daartoe lei dat 'n vinnige en effektiewe dog amper-seker suboptimale beplanner vergelykbare of beter resultate kan behaal as die van 'n asimptotiese optimale beplanner. Resultate van eksperimente dui aan dat dit wel die geval is.

Sonder om te demonstreer dat die beplande paaie inderdaad uitvoerbaar is, is die praktiese nut van die ontwikkelde geoptimeerde padbeplanning-algoritmes onseker. 'n Padvolger is dus ontwikkel wat beplande paaie akkuraat volg in die afwesigheid van steurseine, en dit is ook in staat om afwykings te korrigeer wanneer steurseine ondervind word.

Acknowledgements

“It always seems impossible until it is done.” – Nelson Rolihlahla Mandela.

This research has been the biggest, most challenging, and yet most rewarding adventure in my life thus far. Looking back, I have no regrets for leaving gainful employment on its account – I believe the rewards greatly surpass the losses. I strongly believe that I would have not made it through this masters research, to the point of handing in this thesis, without the support of the great people around me that has carried me through it all. I therefore take this opportunity to sincerely thank each of these awesome people for their support.

I have no words to express my gratitude to my supervisors, Dr. Corné van Daalen and Mr. Johann Treunicht, for giving me the opportunity to pursue this research under their able supervision. I know this might sound like a fairytale, but I always wanted to be Dr. van Daalen’s student and do the cool stuff that he does from the first day I landed on the ESL webpage after completing my BEng degree at the University of Swaziland. Thank you Dr for your warm welcome from that very day and for your positive prospects for the possibility of that aspiration. Great thanks to Mr. Treunicht who offered me the bursary to work on this project two years later and who, without me asking, nominated Dr. van Daalen to co-supervise and be the primary supervisor of my project. Most importantly, I would like to thank both of them for their invaluable guidance, motivation and support throughout the duration of this research. I have indeed been standing on shoulders of giants.

Great thanks also to Sandile Mkhali (also known as Mr. Sharp), my fellow countryman, undergraduate classmate, masters lab-mate and probable future best man, for sharing the bursary advert with me. I would not have known of this opportunity, let alone working on this project, if you did not dare to share this information. Special thanks go to one of the coolest lecturers I have ever known, Dr. Japie Engelbrecht, for his input in expediting the processing of documents required for my visa application prior to the start of my first year. Thank you also for your constructive feedback during the course of my project and for constantly reassuring me that the end does come eventually – indeed it has come. I cannot help but admire your great sense of humour, which made me realise that being in academia does not make one a villain after all. Thanks also to Dr. Willem Jordaan, who used to call me “Bongi”, and who constantly, and with genuine interest, kept on checking up on me throughout my time at the ESL.

How can I forget to thank the rest of the robotic people at the ESL for their warm welcome and their support throughout this bumpy, but rewarding, research journey? Thank you all for making my stay pleasant, and for helping me gain a broader understanding of robotics through your different projects. I wish you all the best in your future endeavours. I would also like to specially thank the administration of the lab for keeping the periodic lab braais and camps alive. Thank you for the free food, it was really appreciated! On that note, apologies for the few occasions where I remained behind, glued on my computer screen – acting like a nerd, when you guys went out. Still in the spirit of free stuff, I would like to thank the coffee machine administrator, Clint Lombard, for allowing me to default on payments when I did not have money, but still get my much-needed cups of coffee. God bless you sir! Thanks also to my officemates, Dinorego Mphogo and Joshua Mfiri, from whom I learnt a lot, not only about their perspectives on control systems and artificial intelligence, but also about deep life and survival principles. Particular thanks to Dino for always bringing home-made food for us to devour in the office. You really made our stay pleasant.

The work presented in this thesis was performed in a number of places, most notably: three

apartments I rented during my stay at Stellenbosch, my allocated office at the ESL, the Carnegie research commons at the J. S. Gericke library of Stellenbosch University, at home in Mpaka (Eswatini), on flight while travelling to Stellenbosch from home and vice versa, at the University of Johannesburg while on private visits, and at the University of the Witwatersrand while attending the Deep Learning Indaba conference. I would like to convey great thanks to everyone who made the environment conducive in these places for me to be able to conduct my work. To my three landlords at Stellenbosch, namely the Christians family, the Bennet family, and the Anolds family, I am grateful for making my stay in Stellenbosch feel like home. Thanks for every meal and moments of laughter you shared with me. I am grateful to Stellenbosch University for letting me use their world-class facilities, both in the ESL and at the Carnegie research commons. To my mom, who prepared homely meals and offered motherly support in all occasions I worked from home, I cannot thank you enough. Thanks also to every pilot who manned every aircraft I boarded for the smooth flights, which enabled me to do some of my work above the clouds! Great thanks to Abiola Bolaji who provided a very hospitable environment when I visited at the University of Johannesburg, allowing me to do a considerable portion of my work there. To the organisers of the Deep Learning Indaba conference, I am grateful for sponsoring my attendance of the conference, the opportunities to network with leading researchers in AI, the opportunity to present my work, the feedback I received and the prize that you awarded me. I appreciate everyone who did not hold back questions during the various presentations of the work contained in this thesis. Every question helped me rethink some aspects of the project and as such, each question reshaped this thesis in its own special way.

Now, allow me to send my word of appreciation to the people who selflessly, and without expecting anything in return, helped me by reviewing the content of this thesis whenever I asked. Great thanks to both my supervisors for their informed, detailed and insightful reviews. I am also grateful to Feziwe Mamba, who did not mind sacrificing her time on numerous occasions for this purpose, when she would have otherwise spent it doing her own masters work in polymer science. Sandile Mkhalihi has also been instrumental in this regard, and for that I am very grateful. I highly value the feedback I received from my mom, who was the first person to whom I demonstrated the work presented in this thesis in the form of simulations and videos for the different developed algorithms as well as the integrated system. Thank you all, without your reviews this thesis would have not been the same.

Hearty thanks to all my fellow Swati people who were with me on “national duty” at Stellenbosch, with whom I would converse in my home language, making Stellenbosch feel like home away from home. These are: Dr. Nathi Gule, Feziwe Mamba, Sandile Mkhalihi, Nomcebo Masilela, Ziningi Mathabela, Zinhle Dlamini, Temakholo Mathunjwa, Nosisa, Mfanelo, Lindo and Msizi. You guys are awesome. Thanks also to my Zulu, Ndebele, Xhosa, Pedi, Sotho, Tsonga, Venda and Tswana people, with whom I would engage in flawless conversations, with each of us speaking in their home language – indeed, borders in Southern Africa are only physical! They include: Zimkhitha Sijovu, Makhetha Phofoolo, Axolile Sekobi, Cleoprata Saul, Bonang Maja, Nomfundo Xulu, Thobeka Mhlongwe, Nompumelelo Nyembe, Uvile Asekun, Busisiwe Msutwana, Dawn Mahlangu, Windy Mokuwe, Zintle Magazi, Ndumiso Sibanda, Justice Mukovhe Mukwevho and Asanda Zindela. To my Namibian friend, Mutangeni Iipinge, I am truly grateful for all the pep talks we had. They played a very important role in keeping me focused on my work in spite of all the challenges I encountered along the way. Special thanks to my pastor, Dr. Funlola Olojede, and entire Redeemed Christian Church of God (Desire of Nations parish) crew, who supported me socially and spiritually during my stay in Stellenbosch. Thanks to God Almighty, the Lord Most High, for answered prayers.

I wish to also extend my sincere gratitude to my high school mathematics and science teachers for instilling in me the love for mathematics and science, which has been, without doubt, key in grasping the concepts of path planning, path optimisation as well as path tracking, and implementing them in this project. Special mention to Miss Phindile Ginindza, Madam Nkambule, Miss Nozipho Dlamini and Mr Vusi Sibandze, who did not mind my numerous consultations whenever I needed clarity. In the same vein, I would like to send my sincere thanks to my undergraduate lecturers from whose lectures I was inspired to pursue research in autonomous navigation. These are: Dr. Hidaia Alassouli, who taught me control engineering, and Dr. Armstrong Mazinyane Nhlabatsi, who taught me programming techniques (C and C++), digital systems, as well as mi-

coprocessor and microcomputer systems. Dr. Nhlabatsi also supervised my final year project and it was through that experience that my interest for robotics and artificial intelligence was really ignited. His lectures on programming techniques as well as his approach to tackling complex problems have really helped me a great deal in developing all the algorithms contained in this thesis. I would like to also thank him for his invaluable advise as my personal mentor, which has kept me afloat upto this end in my academic endeavours and in life in general. Dr. Alassoulli's teachings on control systems have served as a great foundation for understanding the control task of path tracking, the associated literature and the subsequent development of the path-tracking algorithm.

Special thanks to the Deitels (Harvey and Paul), the authors of my favourite programming book, *C: How to Program*, which has remained useful to me from the second year of my undergraduate study to date, and has been an immensely incredible reference when implementing algorithms developed in C++ in this thesis. To all authors of books and research papers I have used in this research, thank you for laying the foundation. In the same vein, I would like to appreciate the efforts of everyone who contributed in the development of all software (both open-source and licensed) and hardware tools that I have used in this project.

Now, to everyone who came to lift me up at times when I was down during the course of the project, thanks for having my back. In that regard, I would have erred if I do not specifically mention my longest-serving friend, Mzwandile Tsabedze, who has been an unfailing pillar in many practical ways. He has truly demonstrated true friendship and brotherly love as described by the writing in the Holy book, which says: "A friend loveth at all times, and a brother is born for adversity". Great thanks to the most phenomenal motivational speaker, Dr. Eric Thomas, whose life story and online motivational content kept motivating me and rekindling my hope during hard times.

The English adage says "blood is thicker than water". On that note, I would like to thank my family for their incomparable support to this end. To my parents Mrs Nikiwe Carol Maseko (nee Banda) and Mr Clement Mfanawemphi Maseko, to whom I truly belong, I am especially grateful for sincerely giving me your blessings to quit gainful permanent employment and focus on this masters thesis. Then to my siblings: Simphiwe, Bongiswa and Busiswa, thank you for holding the fort back home all this time that I have been away as your elder. Particular thanks to the young man, Busiswa, for his perpetual requests to play various games with him when I worked from home. The numerous unintended breaks I took to honour these requests helped me a great deal to refresh and replenish my strength at times when I would have not done so on my own accord.

Thank you all, the success of this research truly belongs to all of you.

Dedications

This thesis is dedicated to:

- *my parents Mr. Mfanawemphi Clement Maseko and Mrs Nikiwe Carol Maseko nee Banda, who prioritized my education above numerous other needs; I am because they are. I am forever grateful.*
- *my future wife (the bone of my bones), kids and descendants.*

Contents

Abstract	ii
Opsomming	iii
Acknowledgements	iv
Dedications	vii
Contents	viii
List of Figures	xii
List of Tables	xv
Nomenclature	xvi
1 Introduction	1
1.1 Background and Motivation	1
1.2 The Autonomous Navigation Framework and Terminology	2
1.3 Targeted Vehicle Class and the Test Vehicle	5
1.3.1 Targeted Vehicle Class: Vehicles with Constrained Kinematics	6
1.3.2 Test Vehicle	6
1.4 Project Overview: Proposed Architecture, Assumptions, Objectives and Contributions	6
1.4.1 Proposed System Architecture	6
1.4.2 Assumptions	8
1.4.3 Project Objectives	8
1.4.4 Contributions	9
1.5 Thesis Organisation	9
I Literature Review	11
2 Optimised Path Planning	12
2.1 Overview	12
2.2 Definition of Path Planning	13
2.3 Path Planning Concepts	13
2.4 Combinatorial Path Planning	15
2.4.1 Properties of Combinatorial Path Planners	16
2.4.2 Visibility Graphs	16
2.4.2.1 Conclusion on Visibility Graphs	18
2.4.3 Generalised Voronoi Diagrams	19
2.4.3.1 Conclusion on Generalised Voronoi Diagrams	19
2.4.4 Cell Decomposition	19
2.4.4.1 Vertical Cell Decomposition	20
2.4.4.2 Triangulation	21
2.4.4.3 Conclusion on Cell Decomposition	21

2.4.5	Conclusion on Combinatorial Path-planning Techniques	22
2.5	Grid-based Path Planning	23
2.5.1	Assumptive Path-planning in Grid Costmaps	24
2.5.1.1	Fundamentals of Grid-based Path-planning using Graph Search	25
2.5.1.2	Optimal Path-planning in Grid Costmaps	25
2.5.1.2.1	Dijkstra's algorithm	25
2.5.1.2.2	A* Search	26
2.5.1.3	Incremental Replanning in Grid Costmaps	28
2.5.1.4	Anytime Path-planning in Grid Costmaps	30
2.5.1.4.1	A* with an Inflated Heuristic	30
2.5.1.4.2	Anytime A*	31
2.5.1.4.3	Anytime Repairing A*	32
2.5.1.5	Anytime Replanning in Grid Costmaps	34
2.5.1.6	Conclusion on Assumptive Path-planning in Grid Costmaps	35
2.5.2	Grid-based Path-planning Under Uncertainty	35
2.5.2.1	Conclusion on Grid-based Path-planning Under Uncertainty	38
2.5.3	Conclusion on Grid-based Path Planning	38
2.6	Sampling-based Path Planning	38
2.6.1	Fundamentals of Sampling-based Path Planning	39
2.6.2	Single-query Sampling-based Feasible Path Planning	41
2.6.3	Single-query Sampling-based Anytime Path Planning	45
2.6.4	Single-query Sampling-based Optimal Path Planning	48
2.6.4.1	The Optimal RRT (RRT*)	49
2.6.4.2	Notable Successors of the RRT*	50
2.6.4.3	The Informed RRT*	51
2.6.4.4	The Wrapping-based Informed RRT*	54
2.6.5	Single-query Sampling-based Replanning	56
2.6.6	Conclusion on Sampling-based Path Planning	57
2.7	Local Path Planning	59
2.7.1	Dubins Local Path-planning Method	59
2.7.2	Reeds-Shepp Local Path-planning Method	60
2.7.3	Continuous-curvature Local Path-planning Method	61
2.7.4	Conclusion on Local Path-planning	62
2.8	Path Optimisation	62
2.8.1	Shortcut-based Path Optimisation	63
2.8.1.1	Path Pruning	63
2.8.1.2	Random Shortcut Method	64
2.8.1.3	Wrapping Process	65
2.8.1.4	Conclusion on Shortcut-based Path Optimisation	65
2.8.2	Gradient-based Path Optimisation	66
2.8.2.1	Conclusion on Gradient-based Path Optimisation	70
2.8.3	Conclusion on Path-optimisation Techniques	70
3	Path Tracking	72
3.1	Elements of a Path Tracking Algorithm	72
3.2	Geometric Path Trackers	74
3.2.1	Head-to-goal Path Tracker	74
3.2.2	Follow-the-carrot Path Tracker	75
3.2.3	Pure-pursuit Path Tracker	76
3.2.4	Vector-pursuit Path Tracker	77
3.2.5	Continuous-curvature Path Tracker	77
3.3	Path Tracking with Position-based Feedback	78
3.3.1	Path Tracking with Rear-wheel Position-based Feedback	78
3.3.2	Path Tracking with Front-wheel Position-based Feedback	80
3.4	Conclusion on Reviewed Path Tracking Techniques	81

II Development of Algorithms, Analysis and Results	83
4 Development of the Local Path Planner	84
4.1 The Context of the Local Path Planner	84
4.2 Local Path-planning Algorithm Development	85
4.2.1 The Concept of a Manoeuvre-based Local Path Planner	86
4.2.2 Local Path-planning Query Specification: Implications for the Local Path Planner	87
4.2.3 Development of the Local Path Planners	88
4.2.3.1 Motion Primitives for Continuous-curvature Paths	88
4.2.3.2 Continuous-curvature Local Path Planner with Standard Inputs	90
4.2.3.2.1 Development of Dubins LPM with Standard Inputs	90
4.2.3.2.2 Results from Runs of Developed Dubins LPM with Standard Inputs	94
4.2.3.2.3 Adaptation of Dubins LPM with Standard Inputs to the CC LPM with Standard Inputs	95
4.2.3.2.4 Results from Runs of the Developed CC LPM with Standard Inputs	102
4.2.3.3 Continuous-curvature LPM with “Don’t Care” Goal Orientation	103
4.2.3.3.1 Development of Dubins LPM with “Don’t Care” Goal Orientation	104
4.2.3.3.2 Results from Runs of Developed Dubins LPM with “Don’t Care” Goal Orientation	106
4.2.3.3.3 Development of the Continuous-curvature LPM with “Don’t Care” Goal Orientation	106
4.2.3.3.4 Results from Runs of the Developed CC LPM with “Don’t Care” Goal Configuration	107
4.2.4 Local Path-planning Method Summary	109
5 Development of Optimised Global Path Planning Algorithms	111
5.1 Acquiring Inputs for the Global Path Planner	111
5.1.1 Robotic Map Representations: Selecting a Map Representation	112
5.2 The Optimised Global Path-planning Approach	114
5.3 Algorithm Development: Notation and Problem Formulation	116
5.4 Algorithm Development: Adaptation of the Anytime RRT and the Informed RRT* for Continuous-curvature Path Planning	117
5.4.1 Primitive Procedures and the Continuous-curvature Basic RRT	117
5.4.2 Continuous-curvature Anytime RRT	120
5.4.3 Continuous-curvature Informed RRT*	124
5.4.4 Summary: Adaptation of the Anytime RRT and the Informed RRT* to the Continuous-curvature Case	129
5.5 Algorithm Development: Continuous-curvature Path-optimisation Algorithms	129
5.5.1 Continuous-curvature Shortcut-based Path Optimisation	130
5.5.1.1 Continuous-curvature Path-pruning Algorithm	131
5.5.1.2 Continuous-curvature Random-shortcut Method	132
5.5.1.3 Continuous-curvature Wrapping Process	133
5.5.1.4 Summary: Developed Shortcut-based Continuous Curvature Path Optimisation Algorithms	136
5.5.2 Continuous-curvature Gradient-based Path Optimisation	136
5.5.2.1 Summary: Developed Gradient-based Path-optimisation Algorithm	138
5.6 Optimised Path Planning Results: Incorporating Path Optimisation into the Benchmark Path Planning Algorithms	139
5.6.1 Experimental Approach	140
5.6.2 Analyses and Comparison of the Benchmark Global Path Planners	140
5.6.3 Analyses and Comparison of the Optimised Global Path Planners	146
5.6.3.1 Optimised Path Planners based on Path Pruning	146
5.6.3.2 Optimised Path Planners based on Random Shortcut	150
5.6.3.3 Optimised Path Planners based on the Wrapping Process	155
5.6.3.4 Optimised Path Planners based on Gradient-based Path Optimisation	159

5.7	Optimised Global Path Planning Summary	163
6	Path-tracking Controller Design and Implementation	165
6.1	Plant Modelling	165
6.1.1	Axis Systems	165
6.1.1.1	Inertial Axes	166
6.1.1.2	Body Axis	166
6.1.1.3	Coordinate Representation and Transformations	167
6.1.2	Robot Dynamics and Robot State Estimates	167
6.1.2.1	Robot Dynamics	167
6.1.2.2	Kinematic Model	169
6.1.2.3	State Estimates	170
6.2	Path-tracking Controller Design	170
6.2.1	Feed-forward Control	171
6.2.2	Cross-track Error Control	171
6.3	Path-tracking Controller Summary	179
III	Integrated System Results and Conclusions	180
7	System Integration, Autonomous Navigation Experiments and Results	181
7.1	Robot Platform	181
7.1.1	Robot Hardware and Firmware	181
7.1.2	Robot Software	183
7.1.2.1	The Robot Operating System (ROS) and ROSARIA	186
7.2	The Simulated Robot Platform	188
7.2.1	Desirable Features of a Simulation Environment	189
7.2.2	Considered Simulation Environments	189
7.2.2.1	MobileSim	189
7.2.2.2	V-REP	190
7.2.2.3	Gazebo	191
7.2.2.4	The Simulator of Choice	193
7.3	Overview of the Integrated System	193
7.4	Deployment of the Integrated System through the ROS Navigation Stack	194
7.5	Autonomous Navigation Experiments and Results	198
7.5.1	Path Tracking in the Absence of Disturbances	199
7.5.2	Path Tracking in the Presence of Disturbances	199
7.5.3	Summary: Autonomous Navigation Experiments and Results	201
7.6	Chapter Summary	202
8	Conclusion and Future Work	206
8.1	Summary and Contributions	206
8.1.1	Summary	206
8.1.2	Contributions	208
8.2	Future work	209
	Appendices	210
A	Analyses of Optimised Path Planners	211
A.1	Confidence Intervals	211
	Bibliography	213

List of Figures

1.1	The ESL AutoNav framework (adapted from Van Daalen's PhD thesis [22])	3
1.2	The autonomous navigation problem	3
1.3	Proposed system architecture	7
2.1	An example 3D workspace	14
2.2	Workspace and configuration space comparison	15
2.3	Illustration of the visibility graph path-planning technique	17
2.4	Selection of edges for the reduced visibility graph	18
2.5	The reduced visibility graph path-planning technique	18
2.6	Maximum-clearance path going between two straight-line obstacle edges	19
2.7	Path planning using Voronoi graphs	20
2.8	An illustration of the vertical cell decomposition method for path planning	21
2.9	An illustration of the triangulation method for path planning using cell decomposition	22
2.10	Configuration space discretisation for grid-based path planning	24
2.11	An illustration of the sampling-based path-planning approach	40
2.12	An illustration of the basic RRT path-planning algorithm	43
2.13	The region sampled by the informed RRT*	52
2.14	Example runs of the RRT* and the informed RRT* for the same path planning problem.	53
2.15	An illustration of the wrapping process used by the wrapping-based informed RRT*	54
2.16	Example Dubins paths	60
2.17	An illustration of a Dubins car trapped in a narrow cul-de-sac	60
2.18	The shortest Reeds-Shepp path is sometimes shorter than all contending Dubin's paths	61
2.19	An illustration of the application of shortcut-based path optimisation	63
2.20	An illustration of the path pruning technique	64
2.21	An illustration of the random shortcut method	65
2.22	Gradient-based path optimisation illustration	70
3.1	Various ways of describing a path	73
3.2	The geometric bicycle model	73
3.3	The kinematic bicycle model	74
3.4	An illustration of the head-to-goal pursuit-based path tracker	75
3.5	An illustration of the follow-the-carrot pursuit-based path tracker	75
3.6	An illustration of the pure-pursuit path tracker	76
3.7	Continuous-curvature path tracking	78
3.8	Path tracking with rear-wheel position-based feedback	79
3.9	Path tracking with front-wheel position-based feedback	80
4.1	The context of the local path planner within an RRT-based global path planner	85
4.2	An example of a path with a discontinuous curvature profile.	89
4.3	The geometry for the construction of Dubins <i>CCC</i> Paths.	92
4.4	The geometry for the construction of Dubins <i>CSC</i> paths.	92
4.5	Isolation and construction of each Dubins <i>CSC</i> case.	93
4.6	Example Dubins <i>CCC</i> paths generated by our implementation of Dubins LPM.	94
4.7	Example Dubins <i>CSC</i> paths generated by our implementation of Dubins LPM.	95
4.8	An example continuous-curvature (CC) turn.	97

4.9	An example computation of a CC turn with a small deflection.	98
4.10	An example computation of a CC turn with a large deflection.	99
4.11	The geometry for the construction of continuous-curvature <i>RLR</i> and <i>LRL</i> paths . . .	100
4.12	Turning circles for a continuous-curvature local path-planning query.	101
4.13	Depiction of μ -tangent placement for connecting the start-end CC circle pairs.	102
4.14	Example paths, purely made up of turns, generated by our continuous-curvature LPM.	103
4.15	Example continuous-curvature <i>CSC</i> paths generated by our continuous-curvature LPM.	104
4.16	Tangent placement for the construction of the adapted Dubins <i>CS</i> paths.	105
4.17	Geometric construction of Dubins <i>LS</i> and <i>RS</i> paths.	106
4.18	Solution paths generated by our MATLAB implementation of the adapted Dubins LPM that ignores the goal orientation.	107
4.19	Depiction of geometric construction of continuous-curvature paths, with the goal orientation ignored.	108
4.20	Depiction of μ -tangent placement for the construction of <i>LS</i> and <i>RS</i> continuous-curvature paths.	108
4.21	Solution paths generated by our MATLAB implementation of the adapted CC LPM that ignores goal orientation.	109
5.1	An illustration of the difference between metric and topological maps.	113
5.2	The two ways in which a metric map can be expressed.	114
5.3	Example results from six independent runs of the CC basic RRT.	120
5.4	An example run of the CC anytime RRT algorithm.	125
5.4	An example run of the CC anytime RRT algorithm (continued)	126
5.5	An example run of the CC informed RRT*	128
5.6	Examples illustrating the application of the developed continuous-curvature path-pruning algorithm.	132
5.7	Examples illustrating the application of the continuous-curvature random-shortcut algorithm to shorten a path generated by the continuous-curvature basic RRT algorithm.	134
5.8	Examples illustrating the application of the continuous-curvature wrapping process.	135
5.9	An example illustrating the application of our gradient-based path-optimisation algorithm.	137
5.10	A second example illustrating the application of our gradient-based path-optimisation algorithm.	138
5.11	A percentile plot showing the evolution or improvement of path cost over an increasing number of iterations for 400 independent runs of each of the three benchmark global path-planning algorithms.	142
5.12	A plot of the mean path cost per iteration, with a confidence interval, for 400 independent runs of each benchmark global path-planning algorithm.	143
5.13	A percentile plot showing the evolution or improvement of path cost over an increasing number of iterations for optimised path planners based on path pruning	147
5.14	A plot of the mean path cost per iteration, with a confidence interval, for 400 independent runs of each optimised global path-planning algorithm based on path pruning.	148
5.15	A percentile plot showing the evolution or improvement of path cost over an increasing number of iterations for 400 runs of each of the three optimised global path-planning algorithms based on random shortcut.	151
5.16	A plot of the mean path cost per iteration, with a confidence interval, for 400 independent runs of each optimised global path-planning algorithm based on the random shortcut.	152
5.17	A percentile plot showing the evolution or improvement of path cost over an increasing number of iterations for 400 runs of each of the three optimised path planners based on the wrapping process	156
5.18	A plot of the mean path cost per iteration, with a confidence interval, for 400 independent runs of each optimised global path-planning algorithm based on the wrapping process.	157
5.19	A percentile plot showing the evolution or improvement of path cost over an increasing number of iterations for 400 runs of each of the three optimised path planners formed by incorporating gradient-based path optimisation.	160

5.20	A plot of the mean path cost per iteration, with a confidence interval, for 400 independent runs of each optimised global path-planning algorithm based on gradient-based path optimisation.	161
6.1	An illustration of the inertial ENU (east-north-up) and body axis system.	166
6.2	Block diagram for 3DOF dynamics of motion	168
6.3	The kinematic model of the robot.	169
6.4	An example screenshot of the robot's current state.	170
6.5	Architecture of the path-tracking controller.	171
6.6	The model used in this project for the cross-track error control process.	173
6.7	Calculation of the cross-track error and in-track distance for a circular track.	174
6.8	An illustration of the cross-track error rate	175
6.9	Cross-track error controller conceptual block diagram	175
6.10	A step response of the cross-track error model.	176
6.11	Root locus and step response of the cross-track error controller, with single-loop proportional control.	176
6.12	Block diagram of the cross-track error controller	176
6.13	A root locus of the cross-track error controller's inner loop controller.	177
6.14	The root locus and step response of the cross-track error controller's outer-loop controller.	178
6.15	An impulse response of the cross-track error controller	178
7.1	Example picture of the P3AT along with its dimensions. Images obtained from the Pioneer 3AT datasheet [33].	182
7.2	An illustration of the Pioneer family of robot's stand-alone mode.	183
7.3	An illustration of the Pioneer server mode.	183
7.4	ARIA architecture.	184
7.5	Overview of the ArRobot task cycle	185
7.6	Introducing the robot operating system (ROS)	186
7.7	Communication between ROS nodes using topics and services.	187
7.8	Communication between ROS nodes using topics and services with the ROS master.	188
7.9	MobileSim robot simulator with three P3AT robots.	190
7.10	The V-REP simulator	191
7.11	The Gazebo Simulator with a single P3AT robot.	191
7.12	Gazebo and ROS integration.	192
7.13	The integrated system architecture.	194
7.14	The ROS navigation stack	195
7.15	The ESL autonomous navigation framework (shown again for convenience).	196
7.16	The ROS navigation stack in relation to the ESL autonomous navigation framework	196
7.17	Adapting the ROS navigation stack to the ESL AutoNav framework	197
7.18	The generation of the path planned by the C++ implementation of our optimised path planner in RViz	199
7.19	The planned path, with the actual path overlaid (feedback only)	200
7.20	A plot of the cross-track error as a function of time (feedback only)	201
7.21	The planned path, with the actual path overlaid (feed-forward and feedback)	202
7.22	A plot of the cross-track error as a function of time (feed-forward and feedback)	203
7.23	The planned path, with the actual path overlaid (feedback only)	204
7.24	A plot of the cross-track error as a function of time (with disturbances)	205

List of Tables

4.1	Dubins <i>CCC</i> paths and their manoeuvre specifications.	91
4.2	Dubins <i>CSC</i> paths and their manoeuvre specifications.	91
4.3	Possible paths for the adapted version of Dubins LPM with “don’t care” orientation. .	105
5.1	Numerical summary of the statistics of the benchmark path-planning algorithms . . .	144
5.2	Numerical summary of the statistics of the optimised path planners based on path pruning.	149
5.3	Numerical summary of the statistics of the optimised path planners based on random shortcut	153
5.4	Numerical summary of the statistics of optimised path planners based on the wrapping process	158
5.5	Numerical summary of the statistics of the optimised path planners based on gradient-based path optimisation	162
7.1	P3AT specifications	182
A.1	<i>z</i> -values for selected confidence levels.	211

Nomenclature

Acronyms

2D	Two-Dimensional
3D	Three-Dimensional
A*	A-star search algorithm
AD*	Anytime D-star (Anytime Dynamic A-star) search algorithm
AI	Artificial Intelligence
API	Application Interface
ARA*	Anytime Repairing A-star search algorithm
ARCOS	Advanced Robot Control and Operations Software
ARIA	Advanced Robotics Interface for Applications
ARNL	Advanced Robotics Navigation and Localization
ATA*	Anytime A-star search algorithm
AutoNav	Autonomous Navigation
AUV	Autonomous Underwater Vehicles
CC	Continuous-Curvature
<i>CCC</i>	Curve-Curve-Curve path
CHOMP	Covariant Hamiltonian Optimization for Motion Planning
<i>CSC</i>	Curve straining Curve path
D*	D-star (Dynamic A-star) search algorithm
DARPA	Defence Advanced Projects Agency
DCM	Direction Cosine Matrix
DOF	Degrees of freedom
ENU	East-North-Up axis system
ESL	Electronic Systems Laboratory
GVD	Generalised Voronoi Diagram
IRRT*	Informed Rapidly-exploring Random Tree Star
ITOMP	Iterative Trajectory Optimization for Motion Planning
LCQP	Linearly-Constrained Quadratic Program
LPM	Local Path-planning Method
<i>LRL</i>	Left-Right-Left path
<i>LSL</i>	Left-Straight-Left path
<i>LSR</i>	Left-Straight-Right path
MDP	Markov Decision Process
NED	North-East-Down axis system
P3AT	Pioneer 3AT mobile robot
POMDP	Partially Observable Markov Decision Process
PRM	Probabilistic Roadmap

RDT	Rapidly-exploring Dense Tree
RL	Reinforcement Learning
<i>RLR</i>	Right-Left-Right path
ROS	Robot Operating System
RRG	Rapidly-exploring Random Graph
RRT	Rapidly-exploring Random Tree
RRT*	Rapidly-exploring Random Tree Star (Asymptotically Optimal Rapidly-exploring Random Tree)
<i>RSL</i>	Right-Straight-Left path
<i>RSR</i>	Right-Straight-Right path
SIP	Server Information Packet
SLAM	Simultaneous Localisation and Mapping
STOMP	Stochastic Trajectory Optimization for Motion Planning
SSMR	Skid-Steered Mobile Robot
UAV	Unmanned Aerial Vehicles
UGV	Unmanned Ground Vehicle
USV	Unmanned Surface Vehicles
V-REP	Virtual Robot Experimentation Platform
VSM	Vertex Selection Method
WIRRT*	Wrapping-based Informed Rapidly-exploring Random Tree Star

Coordinate systems (path tracking)

X_E, Y_E, Z_E	Inertial axes
X_B, Y_B, Z_B	Body axes
$X_{\text{guidance}}, Y_{\text{guidance}}, Z_{\text{guidance}}$	Guidance axes
(x_a^E, y_a^E)	The x and y coordinates of entity a in the inertial axes (Chapter 6 only)
(x_a^B, y_a^B)	The x and y coordinates of entity a in the body axes (Chapter 6 only)
$(x_a^{\text{guidance}}, y_a^{\text{guidance}})$	The x and y coordinates of entity a in the guidance axes (Chapter 6 only)
(x_a, y_a)	The x and y coordinates of entity a in the inertial axes (elsewhere, save for Chapter 6)

Symbol Conventions

x	Scalar
$x(t)$	Time-varying vector
\mathbf{x}	Vector
$\mathbf{x}(t)$	Time varying vector
X	Set
$ X $	Number of elements in set X
\mathbf{X}	Matrix
\dot{x}	Derivative of x
\bar{x}	Mean of x
$\ x\ $	The magnitude of x

Subscripts

track	Track (the portion of the planned path connecting two consecutive path waypoints)
-------	---

cTrack	Centre of track
rob	Robot
x	x component of a vector
z	z component of a vector
ffw	Feed-forward
fb	Feedback
c	Correction
ct	Cross-track
e	Error
source	Source waypoint (the waypoint from which the current track starts)
dest	Destination waypoint (the waypoint at which the current track ends)
cte	Cross-track error
ref	Reference
r	Rear wheel
f	Front wheel
rand	Randomly sampled
t	Length of first segment in a Dubins path
u	Length of second segment in a Dubins path
v	Length of third segment in a Dubins path
end	The end of a local path
start	The start of a local path
α, β, γ	Angular distances of the first, second and third turns in a Dubins <i>CCC</i> path
d	Length of the straight-line manoeuvre in a Dubins <i>CSC</i> path
ir	Right-turning circle at initial configuration
il	Left-turning circle at initial configuration
tl	Left-turning tangent circle
tr	Right-turning tangent circle
gr	Right-turning circle at goal configuration
gl	Left-turning circle at goal configuration
soln	Solution

Superscripts

B	Body axes
E	Inertial axes
guidance	Guidance axes

Symbols

c_{best}	Current best cost
c_{best}	The cost of the current best solution
c_{min}	The theoretical minimum cost
c_{new}	Cost of new solution
c_b	Cost bias factor
d_b	Distance bias factor
p	Probability of an event
\mathbf{q}	Configuration (path planning) or waypoint (path tracking)

\mathbf{q}_I	Initial configuration
\mathbf{q}_G	Goal configuration
$\mathbf{q}_{\text{sampled}}$	Sampled configuration
$\mathbf{q}_{\text{nearest}}$	The nearest node
\mathbf{q}_{near}	A near node
\mathbf{q}_{min}	Cheapest among considered neighbours
\mathbf{q}_{new}	New configuration
\mathbf{q}_{cur}	Node from which graph extension is to be attempted (general template for sampling-based path planning)
\mathbf{q}_{rand}	Randomly sampled configuration
$\mathbf{q}_{\text{candidate}}$	Candidate configuraion (pending collision checks)
\mathbf{q}_{temp}	Temporary configuration
$\mathbf{q}_{\text{start}}$	Start configuration of a local path
\mathbf{q}_{end}	End configuration of a local path
\mathbf{Q}	A sequence of configurations that form a path
\mathbf{Q}_{near}	A set of nodes lying in the neighbourhood of a node
\mathbf{Q}_{sol}	A set of solution paths
\mathcal{C}	Configuration space
\mathcal{C}_f	Informed subset
$\mathcal{C}_{\text{free}}$	Free portion of the configuration space
\mathcal{C}_{obs}	Obstacle regions in the configuration space
\mathcal{W}	Workspace
$\mathcal{W}_{\text{free}}$	Free portion of the workspace
\mathcal{WO}_i	The i^{th} obstacle in the workspace
RM	Roadmap
V	Vertex set
E	Edge set
\mathcal{G}	Planning graph
v	A vertex of a planning graph
v_I	Initial vertex (grid-based path planning)
v_G	Goal vertex (grid-based path planning)
$g(v)$	The cost-to-come for vertex v on a planning graph
$h(v)$	The cost-to-go heuristic for vertex v on a planning graph
$f(v)$	The f -value (estimated cost from the root to the goal, via vertex v)
$c(v', v)$	The cost of an edge from vertex v' to the vertex v on a planning graph
R_i	The robot's current position in grid-based replanning algorithms
ϵ	Inflation factor
b	Belief
\mathcal{B}	Belief space
s	State (planning under uncertainty)
π	Policy in POMDPs and Reinforcement Learning
S	A set of states (planning under uncertainty)
A	A set of actions (planning under uncertainty)
O	A set of observations (planning under uncertainty)
b_0	Initial state probability distribution in POMDPs

T	Transition function in POMDPs
Z	Observation function in POMDPs
R	Reward function
γ	Discount factor
V^π	Value function
\mathcal{T}	Tree
r_{\min}	Minimum turning radius
κ	Curvature
α	Sharpness of a clothoid
W	Mahalanobis distance
α	Learning rate or step length
Φ	Constraint matrix
\mathbf{p}	Iterate in gradient-based path optimisation
\mathbf{q}	A vector of configurations forming a solution path
\mathcal{G}	Planning graph
\mathcal{T}	Planning tree
D_1	First discretisation parameter in wrapping-based informed RRT*
D_2	Second discretisation parameter in wrapping-based informed RRT*
u	Control input
\mathcal{M}	Manoeuvre space
\mathcal{U}	Control space
Ω	Centre of a circle
τ	Global path
\mathcal{T}	Set of paths
$\mathcal{T}_{\text{free}}$	The subset of paths that are collision-free
τ^*	Optimal path
m	Map
α	Clothoid sharpness
r_{\min}	Minimum turning radius
\mathbf{F}	Forces
\mathbf{M}	Moments
\mathbf{I}	Moment of inertia
\mathbf{v}	Linear velocity vector
m	Mass
ω	Angular velocity vector
\mathbf{e}	Euler angles
v	Linear velocity in the body x -axis
ω_z	Angular velocity about the body z -axis
θ	orientation of the robot
e_{ct}	Cross-track error
θ_e	Orientation error
\mathbf{q}_d	Desired robot configuration
$\mathbf{q}_{\text{source}}$	Source waypoint
\mathbf{q}_{dest}	Destination waypoint
θ_{track}	Track heading

l_{track}	Length of track
$(x_{\text{cTrack}}, y_{\text{cTrack}}^E)$	Centre of circular track in inertial axes
r_{track}	Radius of circular track
$G(s)$	Plant transfer function
D'	Inner-loop controller
D''	Outer-loop controller
k'	Inner-loop controller's gain
k''	Outer-loop controller's gain
$G_{\text{CL}}(s)$	Closed-loop transfer function
ω_c	Correction angular velocity
$\theta \ \psi \ \phi$	Yaw, pitch and roll
\mathbf{P}	Position vector
Δt	Time interval
κ	Path curvature
$\omega_{z_{\text{FFW}}}$	Feedforward angular velocity command
$\omega_{z_{\text{FB}}}$	Feedback angular velocity command
ω_{z_c}	Correction angular velocity command
$\omega_{z_{\text{current}}}$	Current angular velocity
$e(t)$	Outer-loop controller error signal
$r(t)$	Outer-loop controller reference signal
$\tan(\delta)$	Steering angle
L	Wheelbase
R	Turning radius
l_d	Lookahead distance
T	Time period
s	Arc length parameter
θ_t	Orientation of path tangent
v_r	Linear velocity of rear wheel
v_f	Linear velocity of front wheel

Chapter 1

Introduction

1.1 Background and Motivation

The idea of autonomous (otherwise known as unmanned, robotic, driverless or self-driving) vehicles predates the invention of the automobile (late 1885), with the former's earliest account being the *da Vinci's Self-Propelled Cart* (1478), which could move such that it follows a predetermined path without being pulled or pushed [1, 2]. Interest in the pursuit for autonomous navigation technology has continued to increase progressively in the automobile era, resulting in systems of increasing sophistication. A remarkable acceleration of progress in the field has only been witnessed recently, with the inception of autonomous vehicles competitions sponsored by the United States of America's Defence Advanced Projects Agency (DARPA) [6]. Interest from automakers and tech giants such as Tesla, Volkswagen, Mercedes, Uber and Google, as well start-ups including nuTonomy and Drive.ai [1, 3, 4], serves as a sign for this progress. Most importantly, approval for autonomous driving in public roads by a number of governments [5], affirms the reality that future transportation systems will be autonomous. Moreover, the introduction of open-source development platforms such as the Robot Operating System (ROS) [7] and realistic simulators such as Gazebo [8], opens up the playground, allowing researchers without access to physical robot platforms to participate in the revolution, and thus promising to further fuel the rate of progress.

Despite all these successes, present-day autonomous vehicles remain semi-autonomous or experimental and a lot still remains to be done towards the realisation of fully autonomous vehicles [9]. For a vehicle to exhibit a fully autonomous behaviour, it must be able to *perceive* ("see", "know" or "learn") its environment, *plan* on how to move within this environment to ensure safety and efficiency and *control itself* to track this plan as closely as possible. This can be loosely referred as a "*sense, plan, act*" framework [10], and it is an iterative process that keeps repeating until the vehicle has achieved its goal.

Two key requirements for such a framework in the context of autonomous navigation are the ability to plan feasible, safe and optimal paths in the presence of obstacles and to execute such paths accurately; deviation from the planned path exposes the vehicle to the risk of collision. These focus areas are called optimised path planning and path tracking respectively in robotics literature. An interesting case is that of vehicles with constrained kinematics, for which the planned path has to satisfy certain kinematic constraints. These constraints include the maximum turning rate and the maximum allowable sharpness, where sharpness refers to the rate of change of curvature and is determined by the speed of the actuator controlling the vehicle's heading. Without the sharpness constraint being taken into account, for a vehicle to accurately follow the resulting path, it would have to stop at every point where a change in the orientation of its wheels is required and reorient its front wheels before proceeding [11, 12]. While a number of works in literature have planned paths for such vehicles without taking the sharpness constraint into consideration in cases where the vehicle is not allowed to stop, our position is that doing so, in a way, defeats the whole purpose of planning. This is because if motion constraints are not respected by the planned path, then there is no assurance that the vehicle will (even in the best case) be able to accurately execute the planned path. It is therefore important to incorporate these motion constraints at planning. In addition to the requirement for motion constraints to be satisfied by the output path, two more important attributes of a path-planning algorithm exist. These attributes are its speed in finding

solutions and the quality of the solutions found. Like almost all computational problems, solving this problem requires striking a balance between optimality and execution speed of the algorithm. An interesting and promising approach for addressing this problem works by formulating a path-planning algorithm so that it is quick to find an initial solution, and then use the remaining planning time to improve the solution [13–20]. This is known as *anytime path planning*. This term is derived from the fact that once an initial solution is found, and while improvement is in progress, then anytime the robot is required to start the navigation task, it can simply pick the best path found thus far and execute it. Furthermore, improvement of the remaining part of the path can continue while the robot moves.

With regard to path tracking, a number of approaches exist. A comprehensive review of these approaches can be found in Snider’s work [21]. Snider notes that none of the approaches is perfect for all cases and that even the most simple of them can perform well in some applications. An understanding of the characteristics of the various methods is thus necessary for selection of a suitable method for a given application. In particular, a good path tracker is one which ensures that in the ideal case, the vehicle tracks the planned path accurately and corrects for any deviation in case of disturbances.

The focus of this thesis is on these two focus areas of robotics, namely optimised path planning and path tracking. In optimised path planning, the project investigates strategies to strike the balance between execution speed and optimality of the planning algorithm by extending ideas that exist in literature. As pointed out earlier, in doing so, motion constraints, particularly maximum turning rate and maximum allowable sharpness, will be incorporated to the planning process to ensure that generated paths are executable by the target vehicle. Secondly, the thesis studies the problem of path tracking and selects an appropriate method for our target vehicle. Algorithms resulting from these investigations are then tested on the open-source development platforms mentioned earlier, namely the Gazebo simulator and the Robot Operating System (ROS). The target vehicle for this thesis is an unmanned ground vehicle (UGV), but the implementations are kept generic for ease of extension to other autonomous vehicle classes including autonomous underwater vehicles (AUVs), unmanned surface vehicles (USVs) and unmanned aerial vehicles (UAVs). The unifying characteristic between these vehicle classes (with the exception of omni-directional ones) is that they all have a constraint (lower bound) in turning radius as well as an upper bound on the sharpness of the path they can follow, which are defining characteristics for the path-planning, path-optimisation and path-tracking algorithms developed in this thesis.

The remainder of this chapter serves as a motivation for this research. In Section 1.2, we begin by introducing a general framework for autonomous navigation of which the algorithms developed in this thesis will form part. This framework was introduced in the PhD thesis of Van Daalen [22] at Stellenbosch University in 2010 and has been successfully used by generations of autonomous navigation researchers at the university’s Electronic Systems Laboratory (ESL) thereafter. We discuss the subsystems of this framework, their internal mechanisms and how they interact to give the desired result of navigation without human intervention. The discussion of the autonomous navigation framework is concluded by highlighting the context of this project with respect to the framework. Specifically, we highlight the modules on which this thesis focuses and the inputs that these modules require from other modules of the framework that are outside the scope of the project. Section 1.3 then introduces the class of vehicles targeted by the system developed in this project and the constraints that they place on the proposed system. In Section 1.4, the proposed system architecture, assumptions made and the objectives of the research are presented. Finally, Section 1.5 presents an overview of the rest of the thesis.

1.2 The Autonomous Navigation Framework and Terminology

The Autonomous Navigation and Mapping (AutoNav) research group within the Electronic Systems Laboratory (ESL) of Stellenbosch University is continually working towards the realisation of fully autonomous vehicles through research and development of underlying algorithms and systems. This project is a contribution towards that effort. Figure 1.1 is a general framework for autonomous navigation, which forms the basis of the work done at the lab.

To get a clear understanding of the working of the framework, let us consider the scenario shown in Figure 1.2. In this scenario, we have an autonomous robot in an unknown, dynamic environment.

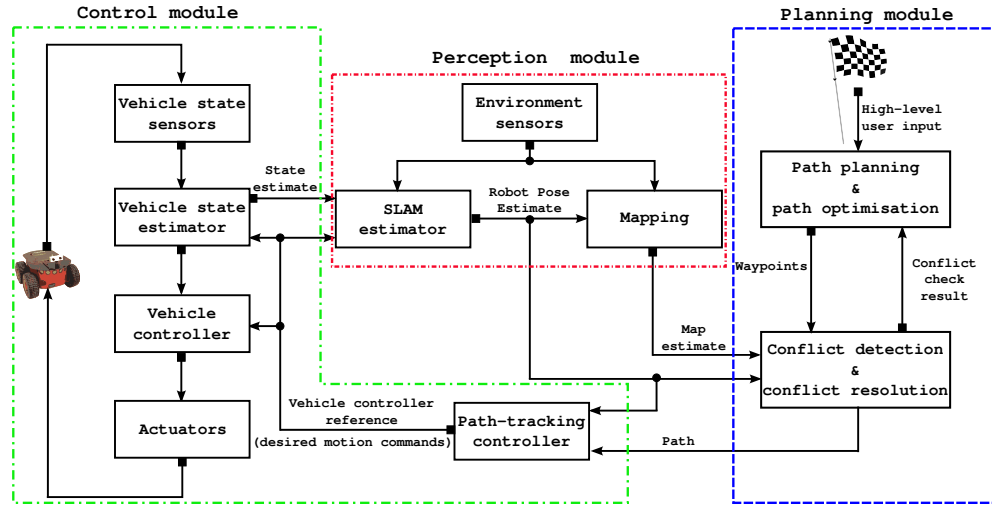


Figure 1.1: The ESL autonomous navigation (AutoNav) framework (adapted from Van Daalen's PhD thesis [22]). Three major modules make up this framework (each marked with a distinct outline).



Figure 1.2: The autonomous navigation problem

We wish to command the vehicle to execute a specific mission. To achieve this, we will first have to communicate mission objectives and mission constraints in the form of a **high-level user input** (which might be from a human user or an artificial decision-making agent) to the **path planning and path optimisation submodule** of the **planning module**. Mission objectives can encompass things like a series of waypoints ($A \rightarrow B \rightarrow C \rightarrow \dots \rightarrow Z$) in the environment that the robot has to visit to accomplish its mission, while mission constraints may include the time by which each location is to be reached, minimum clearance from obstacles, and maximum amount of fuel to be used among others. The path planning and path optimisation submodule examines the user input and uses it to compute a feasible, collision-free and optimal path to be followed by the robot to simultaneously achieve the mission objectives and satisfy mission constraints. To ensure

that the computed path is collision-free, the path planning and path optimisation submodule validates the planned path by passing it through a **conflict detection submodule**. *Conflict* refers to the violation of the required minimum clearance between the robot and other objects in the environment. To classify a path as collision-free or not, the conflict detection submodule makes use of the vehicle's pose (position and orientation) estimate and the environment's map estimate, which are respectively computed by the **SLAM estimator** (simultaneous localisation and mapping estimator) and the **mapping submodule**, which are within the perception module.

The output of the planning module is a collision-free path that can be described using manoeuvres and/or a series of closely-spaced waypoints. These data together with the vehicle's pose estimate are fed to a **path-tracking controller**, which can be seen as a high-level controller. It computes motion commands that minimise the cross-track error and the along-track error to keep the vehicle as close as possible to the planned path. The tracking of these motion commands is a task of a low-level onboard **vehicle controller**.

The functions of the perception module, the planning module, and the control module are separately explained in the following three paragraphs for clarity.

Perception module: The task of the perception module is to compute an estimate of the robot's pose as well as an estimated representation (map) of the environment. It does this through processes respectively known as localisation and mapping. These processes are highly intertwined in that for the robot to accurately localise itself, it needs a high quality map, and to create a good map (mapping), it has to know where it is (localisation). Thus these problems can be viewed as a version of the “*chicken and egg problem*” [23]. This has led to researchers adopting the idea of solving them simultaneously through a process known as simultaneous localisation and mapping (SLAM). In essence, the question posed by the SLAM problem is whether it is possible for a robot to simultaneously build a map of an unknown environment, while computing its pose within this map [24]. It thus estimates both the locations of obstacles and the trajectory of the robot. This information is useful to both the planning module, and the path-tracking controller. The planning module needs both the map of the environment and the robot pose to plan the solution path while the path-tracking controller only requires the robot's pose, which it uses to compute the error between the robot's pose and a desired pose on the planned path, and uses that error signal to compute corrective motion commands that cause the robot to follow the planned path. For these purposes, the two outputs of the perception module, the map estimate and the robot pose estimate, are respectively relayed to the planning module and the path-tracking controller. Inputs required by the perception module include: pre-existing environment maps (if any), vehicle states and environment states from sensors and/or an estimator. It is important to draw a distinction between the two different maps that have been mentioned. One is used by SLAM for localisation and the other one is built by the mapping function of SLAM and then relayed to the planning module where it is used for collision detection. These maps are also of different types, one is known as a *sparse map* and the other one is a *dense map*. SLAM typically uses a sparse map, where the map constitutes locations of *selected landmarks* in the environment. It is *sparse* in the sense that only selected parts of the environment (the landmarks) are mapped. In contrast, a dense map represents every object in the environment. For path planning purposes, sparse maps are usually not sufficient as they are not complete enough to enable collision detection that considers all obstacles in the environment. For this reason, the collision detector typically makes use of dense maps. Durrant-Whyte et al. [24] gives a comprehensive tutorial on the essentials of SLAM.

Planning module: The planning module receives a high-level user input describing a navigation mission to be executed. This input usually consists of a series of goal configurations that the vehicle has to visit and constraints to be satisfied while executing the mission. Its task is to use this information in conjunction with the estimates for the robot pose and environment map received from the perception module to compute a feasible, collision-free path that takes the robot to the desired goal configuration. The planning task is divided into three components, namely, the *global path planner*, the *local path-planning method (LPM)* and a *collision detector*. Loosely speaking, the task of the global path planner is to find a series of intermediary configurations between the robot's initial configuration and a desired goal configuration through which the robot has to go to reach the goal. Once it has found these intermediary configurations, it then makes

a call to the LPM whose task is to connect each configuration pair with a feasible path that satisfies the robot's motion constraints. Each local path is confirmed collision-free via a call to the collision detector. The result is a feasible collision-free path connecting the robot to its desired goal configuration. If optimality is desired, this path can be passed to a *path optimiser* before being relayed to the control module for execution. Additionally, if the robot is operating in an uncertain and/or dynamic environment where portions of the path can be invalidated by new information, the plan can be periodically updated through a process known as *replanning* and these updates relayed to the control module. LaValle, Latombe, Laumond et al. and Choset et al. [25–28] have written highly-regarded books on the subject of robotic path planning.

Control module: The control module is responsible for generating control commands that cause the robot to accurately track the planned path. Two levels of control are usually implemented: a high-level controller known as a path tracker and a low-level controller usually referred as the on-board vehicle controller. The path tracker is composed of feed-forward and feedback controllers. The feed-forward component of the path tracking control signal is generated directly from the input path information, and it would be sufficient to cause the vehicle to follow the planned path in the ideal case where actuators are perfect, the surface is even and the vehicle hardware, such as tyre pressure, are all perfect. However, since this cannot be ensured in the real world, feed-forward control alone is not sufficient, hence we often need to have a feedback component that corrects for any deviation from the planned path due to disturbances. The feed-forward and feedback control signals are combined to give motion commands, which are then fed as a reference to the low-level controller whose task is to generate electrical signals to drive the actuators to achieve the desired motion. To achieve this path-tracking control objective, an understanding of the dynamics of the robot being controlled and principles of control theory is essential. Existing strategies for robotic path-tracking have been comprehensively surveyed by Snider [21]. Useful control theory literature includes books by Franklin and Powell et al. and Stefani et al. [29, 30]. De Luca et al. [31] discusses the use of feedback control for the specific application of car-like robots.

At this juncture, with the different modules of the autonomous navigation framework outlined and their functions explained, it is appropriate to state the context of this thesis with respect to the framework and the assumptions we make. As stated earlier in Section 1.1, the project is focused on the optimised path planning and path tracking sub-problems of the autonomous navigation problem. As such, with respect to the framework, our focus is on the *planning* and *control* modules – we develop algorithms that equip a mobile robot with the ability to plan a path from one location to another in an unknown environment as well as the ability to accurately execute that plan. We assume the existence of a *perception* module, solving the SLAM (simultaneous localisation and mapping) problem, and providing our algorithms with a map of the environment as well as reliable estimates of the robot's pose.

With the context of the project stated, it is important to disambiguate similar robotics terms that apply to the problem we are addressing to remove any confusion. These terms are *path planning vs. trajectory planning* and *path tracking vs. trajectory tracking*. Path planning and path tracking refer to the case where the path is not parametrised in terms of time. In the case where the path is parametrised in terms of time, the equivalent terms are *trajectory planning* and *trajectory tracking* respectively [32]; thus trajectory planning and tracking are special cases of path planning and path tracking. In the next section, we discuss the class of vehicles that this thesis targets, their motion constraints and the implications of those motion constraints for the path planner and the path tracker.

1.3 Targeted Vehicle Class and the Test Vehicle

This section discusses the class of vehicles to which the algorithms developed in this thesis are applicable (Subsection 1.3.1). It also describes the specific vehicle to be used for the deployment and testing of the system to be developed (Subsection 1.3.2).

1.3.1 Targeted Vehicle Class: Vehicles with Constrained Kinematics

The primary aim of this thesis is to generate paths that are feasible for vehicles with a constrained turning rate and a constrained angular acceleration, and to optimise these paths in such a manner as to preserve this feasibility. Such vehicles are characterised by the fact that they cannot change their heading and heading rate instantaneously. This is due to an upper bound on their angular velocity and angular acceleration, respectively resulting in minimum turning radius (or maximum curvature) and maximum sharpness constraints. The sharpness of a path refers to the rate of change of path curvature along the path with respect to path length. These constraints must be satisfied by the path computed by the path planner and optimiser if the vehicle is to be able to execute it.

Constraining the sharpness (rate of change of curvature) of the path followed by the vehicle prevents sudden changes in the turning rate, therefore ensuring continuous curvature. This continuous curvature requirement on the path followed by these vehicles offers a number of benefits. One such benefit is ride comfort which emanates from the fact that the vehicle is not allowed to execute aggressive turning manoeuvres. Another benefit of paths with a continuous-curvature profile is that the control effort remains within reasonable bounds, and these bounds can be artificially imposed on a vehicle. For these reasons, continuous-curvature paths are not only useful to vehicles with a constrained turning rate and sharpness, but also to vehicles without these constraints in applications where these benefits of continuous-curvature paths are desirable.

As stated in Section 1.1, the thesis applies the developed algorithms to autonomous ground vehicles. As such the vehicles considered are those that move in two dimensions.

1.3.2 Test Vehicle

The particular vehicle to be used to demonstrate the system developed in this thesis is a Pioneer 3AT skid-steered mobile robot from MobileRobots Inc. [34]. A picture of this robot in simulation can be seen in Figure 1.2 which illustrated the autonomous navigation problem in the previous section. While it is a known fact that skid-steered mobile robots do not have a constrained angular rate and angular acceleration (as they are capable of turning on the spot), in this project these constraints will be artificially imposed on the vehicle using a software block. The idea is that instead of passing commanded angular velocities directly to the robot, such commanded velocities are passed through the software block which conditions them in a manner that imposes an artificial angular acceleration to the robot.

At this point, we have introduced the autonomous navigation problem and presented a general framework used in our lab for the development of the different modules that work together to form an autonomous navigation system. We then used that framework as a reference to point out the specific modules that this project seeks to implement. Finally, we described the vehicles for which the system to be developed is aimed for and most importantly, we described the vehicle on which the system will be deployed and the constraints it places on the algorithms to be developed. We now proceed to present the proposed system and the objectives it seeks to achieve, alongside the assumptions made.

1.4 Project Overview: Proposed Architecture, Assumptions, Objectives and Contributions

This project aims to equip a mobile robot with the ability to plan collision-free paths, optimise the planned paths and to execute them accurately. In this section, we draw on the information presented in the preceding sections to give a conceptual diagram of the proposed system, the assumptions made as well as the objectives of the project. These are respectively presented in Sections 1.4.1, 1.4.2 and 1.4.3.

1.4.1 Proposed System Architecture

In Section 1.2, we introduced the autonomous navigation framework. This has helped us to clearly state the context of the path-planning, path-optimisation and path-tracking algorithms to be de-

veloped in this project within a typical autonomous navigation system. Most importantly, it has made it possible to understand the interfaces between the algorithms we will develop in this thesis and other parts of an autonomous navigation system that are not part of this project. Section 1.3 has helped us gain a high-level understanding of the motion constraints of the vehicles targeted by the algorithms to be developed. In this subsection, we use that information to outline the architecture of the proposed system. This is shown in Figure 1.3.

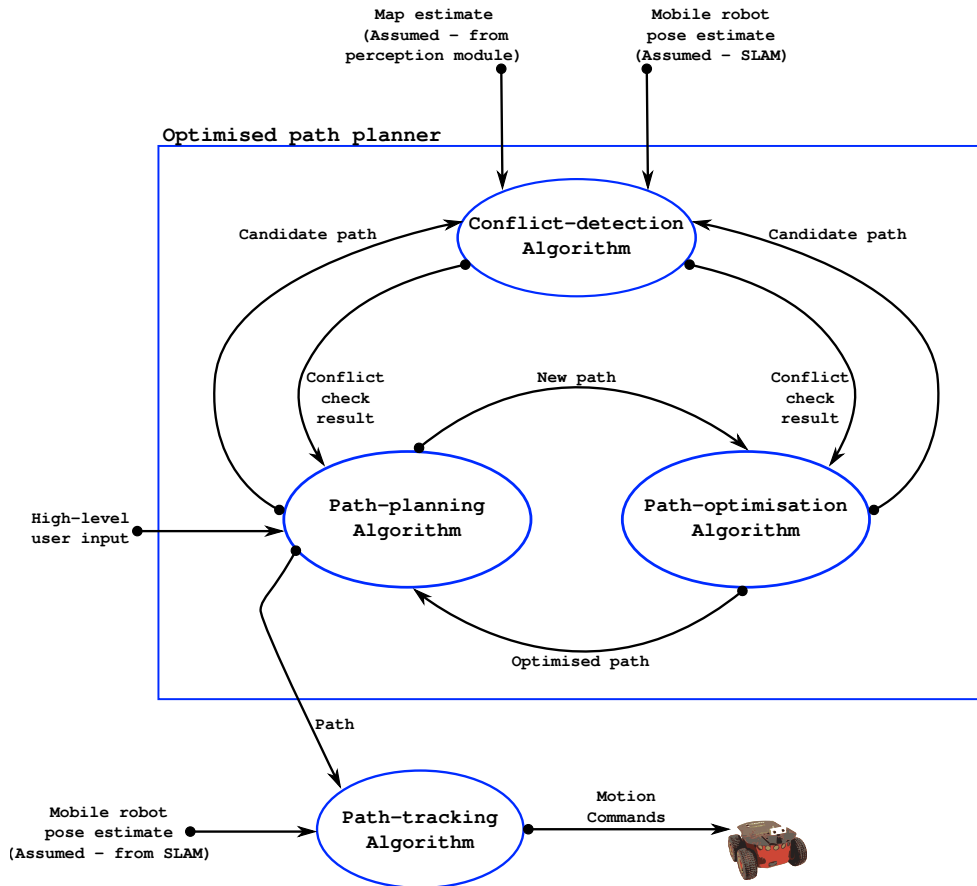


Figure 1.3: The architecture of the proposed system showing the algorithms to be developed in this project, their inputs and outputs as well as the interaction between them.

The system diagram shows the algorithms that will be developed in the project as well as the interaction between them. The inputs required from subsystems of the autonomous navigation framework which are outside the scope of this project are also shown. The proposed system works by receiving a high-level user input describing the autonomous navigation task. In this project, this is a location in the world that the user wishes the robot to visit. A user interface will be provided for the purpose of acquiring this information from the user in the demonstration of the system in Section 7.5. The task is then for our system to find a path that “best” takes the robot from its current location to this target location. Based on the requirements of the user, it might be desired to compute such a path as quickly as possible (quick or efficient planning) or to plan a path that is as short as possible (optimal path planning). This is a path planning task and it is the task of the *path-planning algorithm*. Path-planning algorithms have been presented in literature that find paths quickly (though suboptimal) [35]. On the other end of the spectrum, path-planning algorithms have also been presented that are guaranteed to find optimal paths (though doing so asymptotically) [36]. Quite often, it is desired for the path planner to be both efficient and near-optimal. In such cases, it is necessary to accelerate the path planner’s rate of convergence towards the optimal solution. For this purpose, a *path-optimisation algorithm* can be developed, which

minimises the cost of each new path found by the path-planning algorithm and sends the new solution back to the path planner. The path planner can then use the cost of this new solution to direct its search as it seeks for better solutions. Both the path-planning algorithm and the path-optimisation algorithm make use of a *conflict-detection algorithm* whose task is to classify a given path as being either in conflict with obstacles in the environment or not. Conflict detection is done based on the map of the environment that is assumed to be available from the perception module of the autonomous navigation framework. Collectively, these three algorithms (i.e. the path-planning, path-optimisation and conflict-detection algorithms) constitute the *optimised path planner*. The path found by the optimised path planner is forwarded to the *path-tracking algorithm* whose task is to compute motion commands that cause the robot to follow the planned path. The computation of these desired motion commands is based on the planned path and an estimate of the robot's pose which is assumed to be available from the SLAM submodule of the perception module. Essentially, the path-tracking algorithm computes the error between where the robot is and where it is supposed to be on the planned path, and then uses control engineering techniques to correct this error, thus causing the robot to adhere to the planned path. The result is a system that takes a goal location from a user, plans and optimises a path that leads to that location, and controls the robot to accurately follow this path. In Subsections 1.4.2 and 1.4.3, we explicitly outline the assumptions made and the objectives of this project based on this discussion of the proposed system.

1.4.2 Assumptions

The proposed system architecture of Figure 1.3 makes two assumptions which are important to explicitly point out. These are namely:

- the availability of an accurate localisation system, and
- the availability of an accurate mapping system.

These two submodules which are part of the perception module of the autonomous navigation framework are outside the scope of the project. In this project, we make use of a simulation environment in which it is possible to use existing packages for mapping and localisation.

One more assumption that needs to be stated is the nature of the environments in which the robot will operate. In autonomous navigation, one way in which a robot's environment is usually classified is to classify it based on the dynamics of the obstacles in it. Static environments refer to cases in which all obstacles in the environment are stationary and dynamic environments refer to environments in which the obstacles may move. In this thesis, the robot operates in a static environment. This choice was made due to time constraints and to give priority to the project objectives listed in the following subsection.

1.4.3 Project Objectives

With the proposed system discussed and the assumptions made stated, we are now in a position to outline the objectives that this project seeks to achieve. These are as follows:

1. Selection of suitable path-planning algorithms with diverse optimality guarantees and adapting them for continuous-curvature path planning.
2. Selection of suitable path-optimisation algorithms and their adaptation for continuous-curvature path planning.
3. Investigation of the influence of the selected and adapted path-optimisation algorithms in accelerating the rate of convergence towards the optimal solution among the selected and adapted path-planning algorithms.
4. Developing a path-tracking algorithm that causes the robot to accurately track the planned continuous-curvature paths.
5. Demonstrating the integration of the above-mentioned algorithms into a system capable of autonomous navigation.

1.4.4 Contributions

As a result of the work done in pursuit of the project objectives listed in the preceding section, the following contributions have been made by the thesis:

1. A consolidated review of existing techniques for optimised path planning and path tracking.
2. Adaptation of Dubins local path planner [107] for continuous-curvature local path planning.
3. Adaptation of the basic RRT, the anytime RRT and the informed RRT* path-planning algorithms for continuous-curvature path planning.
4. Analyses and comparison of the performance of the adapted anytime RRT and the informed RRT* path planners in the third point (above).
5. Adaptation of path optimisation algorithms including the path pruning method, the random shortcut method, the wrapping process and a gradient-based path optimisation algorithm, for continuous-curvature path optimisation.
6. Application of the adapted path optimisation algorithms in the fourth point (above) for accelerating the rate of convergence of the anytime RRT and the informed RRT* path planners.
7. Analyses and comparison of the accelerated versions of the anytime RRT and the informed RRT* in the fifth point (above) to study the effectiveness of the different path optimisation algorithms in accelerating their rate of convergence.
8. A path tracking controller using both feed-forward and feedback control to accurately track planned continuous-curvature paths.
9. Adaptation of the ROS navigation stack to the ESL autonomous navigation framework.

1.5 Thesis Organisation

The rest of this document starts off by presenting a detailed review of the relevant literature for each of the sub-problems we seek to address in Part I. The purpose of this literature review is to identify suitable techniques for solving each sub-problem. In Part II, we proceed to develop algorithms addressing each sub-problem based on take-away points from the literature review. The developed algorithms are then integrated to demonstrate autonomous navigation capability in Part III. This last part also evaluates the work done in thesis and suggests directions for future work. The summary of each part of the thesis is given below.

Part I: Literature Review

The first part of the thesis consists of two chapters that review existing literature in the areas of the project. These areas are optimised path planning and path tracking. The literature review is organised such that the main components of the optimised path planner (path planning and path optimisation) are discussed in one chapter (Chapter 2). The discussion of the existing literature for each of these components is wrapped up by a selection of suitable techniques for our project. The second and last chapter of Part I (Chapter 3) is dedicated to literature on path tracking. It is also concluded by a selection of techniques which are found to be suitable for our application. The techniques selected for each problem are carried over to Part II where the implementation of the various algorithms is discussed.

Part II: Development of Algorithms, Analysis and Results

The subject of the second part of the thesis is the development of path-planning, path-optimisation and path-tracking algorithms based on the directions obtained from the literature review. Chapters 4 and 5 discuss the implementation of the optimised path planner. Chapter 4 discusses the development of the local path planning component of the path planner – this is the component

of the path planner which ensures the satisfaction of the vehicle's motion constraints discussed in Section 1.3. Thereafter, Chapter 5 is where the development of the optimised global path-planning algorithm is discussed. This chapter includes global path planning, conflict detection and path optimisation. It also presents an analysis of each of the adapted path-planning and path-optimisation algorithms as well as analysis of the influence of path optimisation in accelerating the rate of convergence of the various selected path planners towards the optimal solution.

Part III: System Integration, Autonomous Navigation Tests and Conclusions

With the algorithms forming the proposed system developed in Part II, Part III focuses on integrating the developed algorithms, demonstration of autonomous navigation capability and conclusions. The opening chapter (Chapter 7) of this part discusses the integration and deployment of the developed algorithms for the purpose of demonstrating the desired goal of autonomous navigation. The robot's hardware and software are introduced in Subsections 7.1 and 7.1.2 respectively. The discussion on the robot hardware helps us understand the physical properties of the robot. The software discussion informs us of the available interfaces for our algorithm to interact with the robot. The discussion continues to introduce the Robot Operating System (ROS) and the benefits of using it in deploying our applications on the robot platform. Building on the discussion on the use of ROS, the following section (Section 7.2) proposes the use of a robot simulator for the demonstration of the integrated system. The choice of the simulator is also discussed. The last part of the chapter (Section 7.5) then presents the autonomous navigation tests and their results. Chapter 8) closes the last part of the thesis (and the thesis itself) by weighing the results of the thesis to its objectives and suggests future research directions.

Part I

Literature Review

Chapter 2

Optimised Path Planning

2.1 Overview

This chapter discusses existing work done by researchers in the global research community to address the optimised path planning problem. Optimised path planning involves not only planning a path from the initial configuration to the goal configuration that is feasible, but one which is also optimal according to some criteria – with path length being the criterion of interest in this project. It is important to make a distinction between optimised path planning and optimal path planning. To reiterate, optimised path planning first generates an initial path and then applies a path-optimisation algorithm to that initial path to get an *optimised path*. On the other hand, optimal path planning does not have an explicit path-optimisation step, but instead, the planning algorithm itself is capable of directly finding the *optimal path*. In optimised path planning, the role of path planning is to find an initial path that possibly has a high cost and the role of path optimisation is to improve this initial solution so as to reduce its cost. Optimised path planning is desirable (over optimal path planning) for the case where planning an initial feasible and collision-free path is far more efficient than directly planning the optimal path [37] – in such instances, most of the allocated time can then be spent improving the initial solution. Also important to note is that some path planners (both optimal and suboptimal ones) strive towards the optimal path by generating a series of solutions where each new solution is used to either limit the search space of the planning problem through bounding the search space using this new solution’s cost or simply using each new solution as the starting point for future searches. In these cases path optimisation can be incorporated into a path planner (either optimal or suboptimal) to accelerate the rate of convergence of the path planner by optimising each new solution, reducing its cost, before it is used to guide future searches.

The objective of this thesis with regard to optimised path planning is two-fold. Firstly, it is to select suitable path-planning algorithms which can be adapted to generate paths that satisfy kinematic motion constraints. It is desired to select a number of such path planners, with diverse optimality guarantees. The intention is to then use them to investigate the effect of a path-optimisation step in accelerating their rate of convergence towards the optimal solution. The second objective is to select suitable path-optimisation algorithms that can be adapted for improving initially-costly continuous-curvature paths. The aim of this literature search is to understand the approaches that have been used previously to solve the problem of optimised path planning, critically evaluate them and identify the appropriate ones in the context of vehicles with a constrained angular velocity and angular acceleration.

The rest of the chapter proceeds as follows: Section 2.2 defines the path planning problem and concepts associated with the problem are introduced in Section 2.3. With the path planning problem defined and the underlying concepts introduced, different path-planning approaches, namely combinatorial, grid-based and sampling-based path planning are then reviewed in Sections 2.4, 2.5 and 2.6 respectively. The discussion of each path-planning approach is concluded by its evaluation according to the objectives of the project and the decision on whether a specific class of path-planning algorithms from that approach has been selected as suitable for the project or not. Section 2.7 discusses local path planning, a component of the sampling-based path-planning approach which was eventually selected as the suitable path-planning approach. Lastly, Section 2.8

reviews existing path-optimisation techniques and selects appropriate ones for the project.

2.2 Definition of Path Planning

As earlier stated in Section 1.2, the path planning task is concerned with computing a path that takes a robot from its initial configuration, \mathbf{q}_I , to a goal configuration, \mathbf{q}_G , while satisfying both the robot's motion constraints and external constraints imposed by obstacles in the environment. A configuration of the robot specifies the position of every point on the robot [28]. Simply put, path planning is about deciding how to get from configuration \mathbf{q}_I to configuration \mathbf{q}_G in spite of the motion constraints. The determination of the goal configuration is the task of a human user or an artificial decision-making agent. The books by LaValle, Latombe, Laumond et al. and Choset et al., respectively titled *Planning Algorithms*, *Robot Motion Planning*, *Robot Motion Planning and Control* and *Principles of Robot Motion: Theory, Algorithms, and Implementation* [25–28] provide comprehensive (and in some instances complementary) discussions on the subject of robotic motion planning of which path planning can be considered as a subset – this is explained shortly. When loosely defining the motion planning problem, Latombe [26] starts by stating it as the problem of deciding the motions that the robot should perform to arrange physical objects in a specified way. This first part of the definition seems to refer to the case of a robot that is assigned with the task of manipulating objects. It is the second part of his definition that resonates well with the path planning problem being addressed in this thesis. In this part, he states that the minimum that could be expected of an autonomous robot is the ability to plan its own motions. According to Guo et al. [38], the motion planning problem can be broken down into two sub-problems, namely, *path planning* and *velocity planning*. Again, path planning plans the path from \mathbf{q}_I to \mathbf{q}_G . Velocity planning then involves planning a velocity profile for the planned path. In this thesis, the velocity planning component of motion planning is fulfilled by planning paths that constitute primitive manoeuvres (i.e. motion primitives) for which corresponding velocities can be easily determined.

With path planning defined, we now proceed to introduce important path planning concepts. The motion planning books listed in the above paragraph as well as research papers on the subject of path planning are used as references in the rest of this chapter, with LaValle's book being the main reference.

2.3 Path Planning Concepts

Before we start reviewing methods that have been used by other researchers to address the path planning problem, it is necessary to understand the inputs to the problem and how they are represented. The basic level at which the context of a path planning problem can be understood is a representation known as the *workspace*, \mathcal{W} . While the workspace represents the path planning problem in a manner that we are most familiar with, most path-planning techniques do not make use of this representation, instead they use a representation known as the *configuration space*, \mathcal{C} . The workspace and the configuration space are introduced next.

The workspace and its representation: The workspace is simply the two-dimensional (\mathbb{R}^2) or three-dimensional (\mathbb{R}^3) Euclidean space in which the robot, denoted by \mathcal{A} , operates. The workspace contains obstacles, with the i^{th} workspace obstacle denoted by \mathcal{WO}_i . It is prohibited for any part of the robot to occupy portions of the workspace occupied by these obstacles – if this happens, a collision will result. The space that is unoccupied by obstacles is known as the *free workspace*; mathematically, it is the set subtraction between \mathcal{W} and the collection of all workspace obstacles, $\sum_i \mathcal{WO}_i$. Figure 2.1 shows an example workspace. An issue with the workspace representation is that the free workspace does not exclusively contain locations that the robot may reach without colliding with obstacles. As the robot moves closer to obstacles in the workspace, there are locations that are part of the free workspace, which, if the robot were to occupy, a collision would occur. This problem is addressed by the configuration space approach which is introduced next.

The configuration space: The idea of the configuration space was introduced by Tomás Lozano-Pérez [25, p. 128] in 1983 [39]. According to LaValle [25, p. 128], the configuration space, \mathcal{C} , is

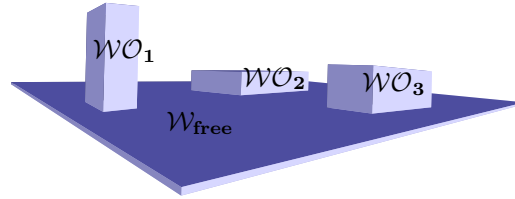
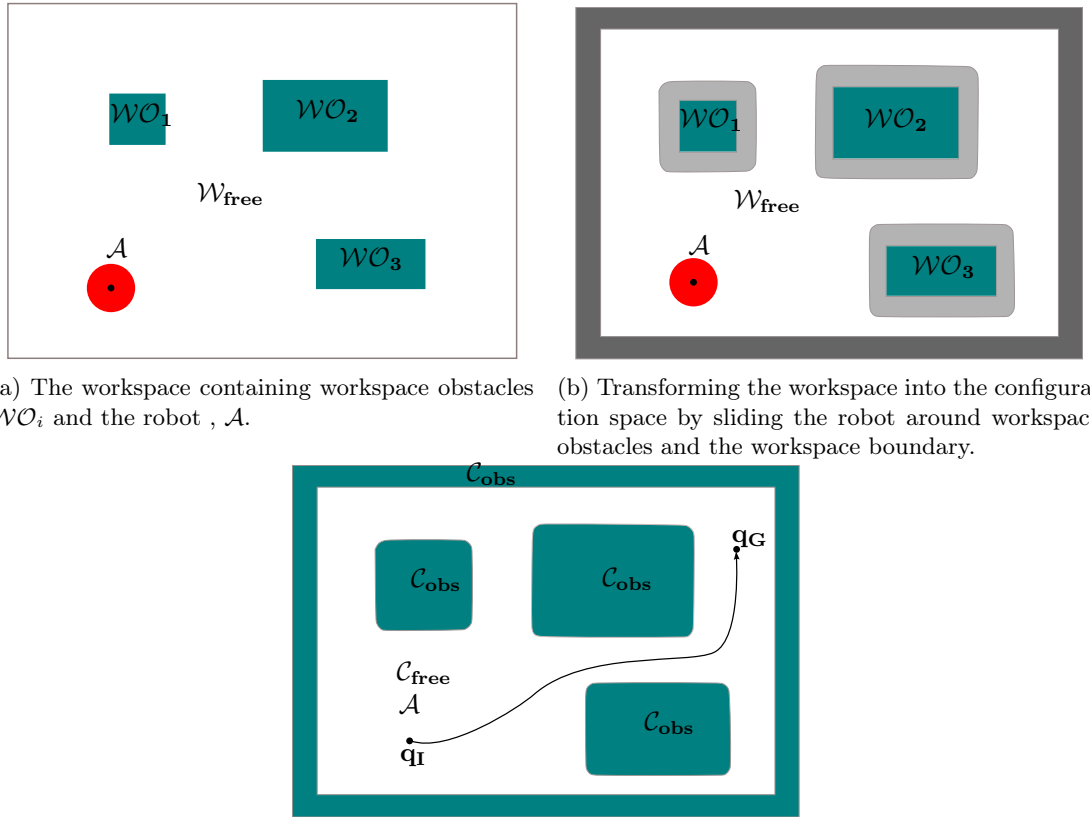


Figure 2.1: An example 3D workspace. This workspace contains three obstacles represented by the 3D boxes and the free workspace is the part that is not occupied by obstacles.

the state space of path planning – a set of all possible transformations that could be applied to the robot given its kinematics. In his configuration space approach to planning [39], Lozano-Pérez states the class of problems that benefit from this approach as those in which it is required to place an object among other objects and to move it without experiencing collisions with the other objects. Clearly, the path planning problem belongs to such a class of problems. The configuration space approach to planning then works by characterising the position and orientation of the object as a single point in the configuration space, where each coordinate represents a degree of freedom in position or orientation of the object – this is called a configuration and it is denoted by \mathbf{q} . The configurations resulting in collision with obstacles are characterised as regions called *configuration space obstacles* and denoted by \mathcal{C}_{obs} . The free space is the part of the configuration space that is not occupied by configuration space obstacles and it is given by $\mathcal{C}_{\text{free}} = \mathcal{C} \setminus \mathcal{C}_{\text{obs}}$, where \setminus denotes a set difference operation. The power of the configuration space representation over the workspace representation comes with the manner in which configuration space obstacles differ from workspace obstacles. Figure 2.2 illustrates this difference by showing how workspace obstacles are transformed into configuration space obstacles. The illustration is done using a circular robot. The procedure works by sliding the robot around each obstacle so that it just touches the obstacle and then tracing the path followed by the centre of the robot. This gives the region in the proximity of the obstacle for which, if occupied by the centre of the robot, a collision will occur. The obstacles are then inflated by this region. This process of inflating obstacles to account for configurations that may cause collisions is known as *Minkowski addition*. In Figure 2.2(b), the region by which obstacles are inflated is represented by the light grey regions surrounding them. The same transformation is applied to the boundaries of the workspace – the dark grey region along the boundary of the workspace denotes the areas where the robot will be at the edge of leaving the workspace. Configuration space obstacles are formed by the inflation of workspace obstacles by the light grey areas as shown in Figure 2.2(c), and the free space, $\mathcal{C}_{\text{free}}$ is the remaining part of the configuration space. With this transformation of the workspace, the configuration space makes it possible to consider the robot as a point mass when planning paths – a very useful simplification. This is because the centre of the robot can now occupy any configuration in $\mathcal{C}_{\text{free}}$ without the robot colliding with obstacles.

With the obstacle region and the free space identified, planning in the configuration space simplifies to searching the free space for a path starting from the initial configuration and ending at the goal configuration. Since the robot is simplified to a point mass, the path is simply a curve in the configuration space as shown in Figure 2.2(c). Different path-planning approaches differ in how they distinguish the obstacle region from the free space and how they search the free space. *Exact* (also called *combinatorial*) approaches explicitly model the obstacle region (or the free space) while *approximate* methods avoid the explicit construction of the free space by relying on a collision detection algorithm which can quickly indicate if a given configuration is in the free space [25, p. 186 and p. 250]. Approximate methods include *grid-based path planners* and *sampling-based path planners*. These path-planning approaches are discussed and evaluated against the project's path planning requirement in the sections that follow. Section 2.4 discusses combinatorial path planning, while Section 2.5 discusses grid-based path planning and Section 2.6 focuses on sampling-based path planning. A planning approach is said to be *complete* if it is guaranteed to find the solution to any planning problem, if one exists; otherwise correctly reporting that there is no solution [25, p.80]. In the mentioned sections of each path-planning approach, completeness of each approach is also discussed. Before proceeding to our review of existing path-planning approaches,



(c) The configuration space, with configuration space obstacles \mathcal{C}_{obs} and free space, \mathcal{C}_{free} . Finding a solution path in the configuration space reduces to searching for a curve that connects the initial configuration, \mathbf{q}_I , to the goal configuration \mathbf{q}_G .

Figure 2.2: Workspace and configuration space comparison.

it is important to reiterate the distinction between two types of path planning, namely *local path planning* and *global path planning*. As stated in Section 1.2, global path planning is concerned with computing a collision-free, feasible path that starts from the initial configuration and ends at the goal configuration (like the one illustrated in Figure 2.2(c)). On the other hand, local path planning computes a feasible path connecting any two configurations in the configuration space, without taking collision constraints into account. Therefore the path-planning approaches that will be discussed next are global path planners – they strive to find a collision-free path that connects the initial configuration to the goal configuration in the configuration space. Local path planning is a component of one of these global path-planning approaches, that is sampling-based path planning – the other global path-planning approaches (combinatorial and grid-based path planning) do not make use of a local path planner. We now discuss the global path-planning approaches, starting with combinatorial path planning.

2.4 Combinatorial Path Planning

Combinatorial path planning is one of the oldest path-planning approaches [40]. In contrast to the other approaches that will be discussed in Sections 2.5 and 2.6, it is exact. As mentioned earlier, this means that it does not approximate the free space, but instead, it uses the geometry of obstacles to construct a *roadmap* in the free space. The roadmap can then be searched for a path connecting any two configurations in the configuration space. To aid the construction of this roadmap, the combinatorial path-planning approach makes the assumption that the shapes and positions of obstacles are known and, in particular, that the obstacles are polygonal. The boundaries (i.e. edges and/or vertices) of the polygonal obstacles are then used to construct the

roadmap. The resulting roadmap is essentially a topological graph in which each node represents a specific configuration in the free space, and each edge represents a straight-line path between neighbouring configurations [28]. As a result, the paths formed are concatenations of straight lines, and as such, discontinuous in curvature. The manner in which the roadmap is constructed differs among different combinatorial path-planning methods. Some methods construct the roadmap directly from the geometry of obstacles while others first use the geometry of obstacles to first break down the free space into cells, and then construct the roadmap based on these cells [25, p. 250]. In particular, combinatorial path planners such as *visibility graphs* and *Voronoi diagrams* construct the roadmap directly from the boundary of obstacles while *cell decomposition* methods such as *vertical cell decomposition* and *triangulation* first use the obstacle boundaries to break the free space into a number of distinct cells, and then create the roadmap using those cells. The different combinatorial path-planning methods are discussed in Subsections 2.4.2, 2.4.3 and 2.4.4.

Once the roadmap is constructed, the task of finding a path from a given initial configuration to a goal configuration (both of which may or may not be part of the roadmap's nodes) reduces to first connecting both the initial and goal configurations to the roadmap, if they are not already connected, and then using classical AI search techniques to search the roadmap for the shortest path connecting the goal configuration to the initial configuration. This will be illustrated in each of the subsections discussing each combinatorial path planner. Before discussing different combinatorial path planners, we first discuss properties that are desirable for planning using roadmaps in the next section – these are the properties that make a roadmap complete.

2.4.1 Properties of Combinatorial Path Planners

For a combinatorial path-planning algorithm to be *complete*, the roadmap (RM) it constructs has to possess three properties, namely: *accessibility*, *departability* and *connectivity* [28]. In essence, accessibility means that given any configuration, \mathbf{q}_1 , which is part of the free space but not part of the roadmap (i.e. $\mathbf{q}_1 \in \mathcal{C}_{\text{free}} \wedge \mathbf{q}_1 \notin RM$), it is possible to find a path from \mathbf{q}_1 to some configuration, \mathbf{q}' which is part of the roadmap (i.e. $\mathbf{q}' \in RM$). Departability has a slightly similar definition to that of accessibility: it says that given any configuration, $\mathbf{q}_2 \in \mathcal{C}_{\text{free}}$, which is not part of the roadmap, it is possible to find a path from some configuration \mathbf{q}'' which is part of the roadmap to \mathbf{q}_2 . Accessibility and departability of a roadmap essentially make it is possible to connect any initial and any goal configuration to the roadmap, as long as they are in $\mathcal{C}_{\text{free}}$. The last desirable property, i.e. connectivity, states that if there exists a path from \mathbf{q}_1 to \mathbf{q}_2 , then there also exists a path from \mathbf{q}' to \mathbf{q}'' on the roadmap – meaning that once \mathbf{q}_1 and \mathbf{q}_2 are connected to the roadmap, it is possible to find a path from \mathbf{q}_1 to \mathbf{q}_2 through the roadmap since it captures the connectivity of $\mathcal{C}_{\text{free}}$. Connectivity guarantees that once the initial and goal configurations are connected to the roadmap, then a path from the initial to the goal configuration can be found through the roadmap.

As long as the roadmap constructed by a combinatorial path planner satisfies the above-discussed properties of accessibility, departability and connectivity, then it is complete. The next subsection discusses the first among the considered combinatorial path planners, visibility graphs.

2.4.2 Visibility Graphs

Visibility graphs, also called *shortest path roadmaps*, create a roadmap in $\mathcal{C}_{\text{free}}$ using the vertices of the polygonal obstacles. Each pair of obstacle vertices is connected with an edge if they are visible to each other. This results in the roadmap formed by the black, dotted lines in Figure 2.3(b) for the path-planning problem shown in Figure 2.3(a). The obstacle vertices which are part of the roadmap form the *vertex set*, V of the roadmap and the edges connecting them form the roadmap's *edge set*, E .

Once the roadmap has been constructed, it can be used to solve any planning query in its domain. Given an initial-goal configuration pair (\mathbf{q}_I and \mathbf{q}_G), the visibility graph method solves the planning query by first adding these configurations to the roadmap as vertices. They are then connected to vertices of the roadmap which are visible to each of them. This is shown in Figure 2.3(c). Once \mathbf{q}_I and \mathbf{q}_G have been added on the roadmap, the shortest path connecting them can be found using a classical AI shortest path finding algorithm such as Dijkstra's algorithm [41] as illustrated by Figure 2.3(d). Since the visibility graph method uses obstacle vertices for the

vertices of its roadmap, the resulting path grazes obstacle vertices as evident in Figure 2.3(d). If it is desired to maintain a certain minimum clearance from obstacles, the obstacles can be inflated by that desired clearance distance prior to the application of the algorithm.

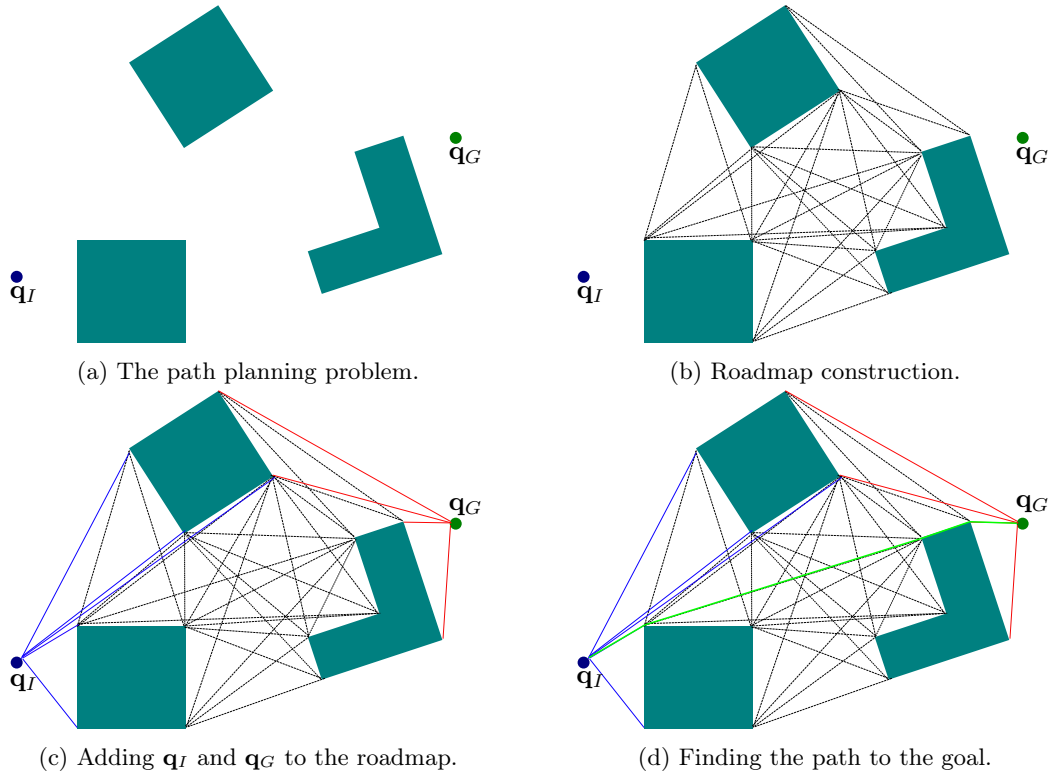


Figure 2.3: An illustration of the visibility graph path-planning technique.

The visibility graph constructed in Figure 2.3 has been constructed using all pairs of visible nodes. It thus contains the maximum number of vertices and edges possible. However some vertices are not very useful and can be removed. The more nodes that can be removed without affecting the completeness of the algorithm, the faster the search for the shortest path. One way to select vertices for removal is to consider each polygonal obstacle vertex for addition to the roadmap's vertex set if and only if it is reflex [25] – a vertex of a polygon is said to be reflex if its interior angle (i.e. the angle that lies in C_{free}) is greater than 180° . As an example, the vertex of the L-shaped obstacle with an interior angle of 90° in Figure 2.3 does not pass this test, and as such, it can be removed. Another way of simplifying the roadmap is to only add edges between a visible obstacle vertex pair if and only if it is possible to extend the edge between them such that the extended line does not poke into C_{obs} at any of the two vertices. In geometric terms, this is best understood via the concept of *supporting* and *separating* lines [28], illustrated in Figure 2.4. A line is called supporting to two obstacles it touches if both obstacles lie on the same side of it. Conversely, if two obstacles lie on different sides of a line that touches both of them, then the line is said to be separating. A line that is neither supporting nor separating pokes into C_{obs} at least at one of the vertices as illustrated in Figure 2.4. It is edges like these that can be discarded in the construction of the simplified version of the visibility graph. This simplified version is known as the *reduced visibility graph*. For the example visibility graph illustrated in Figure 2.3, the corresponding reduced visibility graph is shown in Figure 2.5. With the reduction in number of roadmap vertices and edges, the reduced visibility graph is faster to search for the shortest path than the original visibility graph.

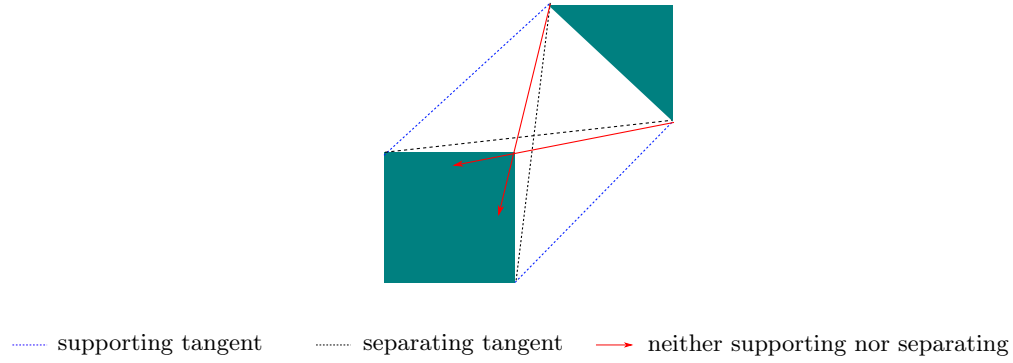


Figure 2.4: An example showing the selection of edges that can be added to the reduced visibility graph. The edges coloured red clearly poke into \mathcal{C}_{obs} at one of the obstacle vertices and can be discarded. Edges that poke into \mathcal{C}_{obs} are those which are neither separating nor supporting.

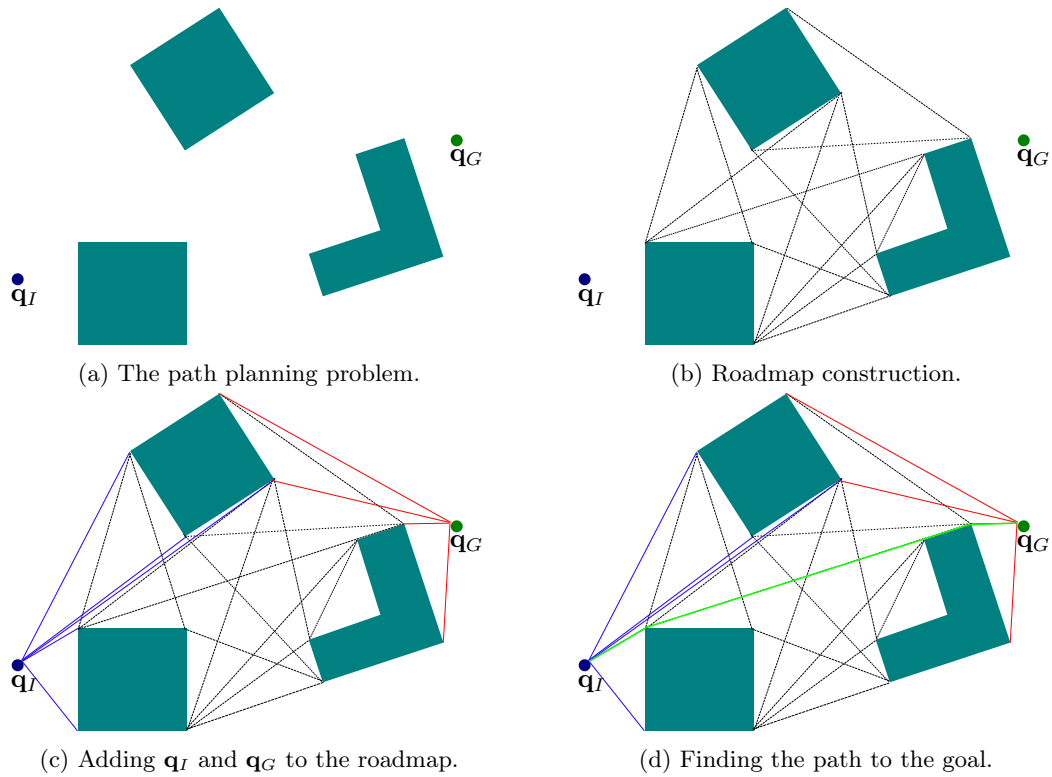


Figure 2.5: An illustration of the reduced visibility graph path-planning technique.

2.4.2.1 Conclusion on Visibility Graphs

In accordance with its other name, i.e. shortest path roadmap, the visibility graph manages to find shorter paths from the initial to the goal configuration. When evaluated against the objectives of this research, however, the planner fails the test. The first reason it fails is because the paths produced are a concatenation of straight lines – hence no continuity in curvature. As such, if such paths are used for a vehicle with a constrained angular rate and angular acceleration, the vehicle will surely fail to follow the path. This would defeat the purpose of planning and accordingly, this planning technique is not selected for our project. The second reason for its unsuitability is its assumption of perfect knowledge of obstacle shape and location, which is unrealistic in the real world. Next, we consider a second combinatorial path-planning technique, generalised Voronoi diagrams (GVDs).

2.4.3 Generalised Voronoi Diagrams

A generalised Voronoi diagram (also known as a *maximum-clearance roadmap* or the *retraction method*) works by building a roadmap that stays as far away as possible from obstacles in the configuration space [25, p. 260]. This is in contrast to the visibility graph path-planning technique discussed in the previous section that constructs paths that tend to graze obstacle vertices. Mobile robotics applications have inherent uncertainties in the pose of the robot, the location of obstacles, as well as in the precision of controllers and actuators. As a result, the built-in headroom in clearance from obstacles associated with Voronoi diagrams is sometimes desirable as it reduces chances of collisions that may occur due to these uncertainties.

Essentially, the GVD is a locus of configurations in the configuration space for which the distance to the two closest obstacles is the same [28]. The method is easily understood with a corridor scenario where the two corridor walls are the obstacles being avoided. This scenario is shown in Figure 2.6 where a path that keeps maximum distance from obstacles is a straight line that runs in the middle of the corridor.

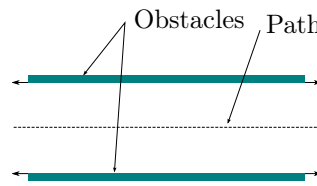


Figure 2.6: The maximum-clearance path going between two straight-line obstacle edges is a straight line. Adapted from LaValle's book [25].

For a given path-planning query, the corresponding Voronoi diagram is constructed by drawing lines of maximum clearance for all obstacle vertex-vertex pairs and edge-edge pairs. For a vertex-edge pair, the maximum clearance curve is a parabola. The roadmap is then obtained by intersecting the curves drawn. Figure 2.7 shows our example for path planning using Voronoi diagrams. The path planning problem being solved is given in Figure 2.7(a). The first step is the construction of the roadmap which is shown in Figure 2.7(b). Once the roadmap is built, the initial and goal configurations are added to the roadmap as shown in Figure 2.7(c). With \mathbf{q}_I and \mathbf{q}_G added to the roadmap, the final step is a graph search for the shortest path leading to the goal as shown in Figure 2.7(d). Again, this search can be done using classical AI shortest path search algorithms.

2.4.3.1 Conclusion on Generalised Voronoi Diagrams

Voronoi diagrams have the desirable feature of keeping maximum clearance from obstacles. However, this desirable feature comes at the expense of optimality in terms of path length – the implication is that the paths produced are surely suboptimal with regards to path length. Moreover, the paths produced are not continuous in curvature – though these paths sometimes have parabolic components, such components do not offer curvature continuity on the overall path as concatenation of straight lines is still possible. Besides, the curvature profile of the parabolas in the Voronoi diagram of a given path-planning query is not guaranteed to be within the vehicle's angular rate and angular acceleration capabilities. Consequently, when weighed against the objectives of this research, this path-planning technique fails to fulfil our requirements.

2.4.4 Cell Decomposition

As mentioned in the introductory part of this section, the cell decomposition method differs from the other combinatorial path-planning techniques in that it does not build the planning roadmap directly from obstacle boundaries (vertices and edges). Instead, this technique uses obstacle boundaries to first break down C_{free} into small regions called *cells*. It is from these cells that the roadmap is then created. The breaking down of the environment into these distinct cells is called *decomposition*. There are three desirable properties that a decomposition technique must possess [25].

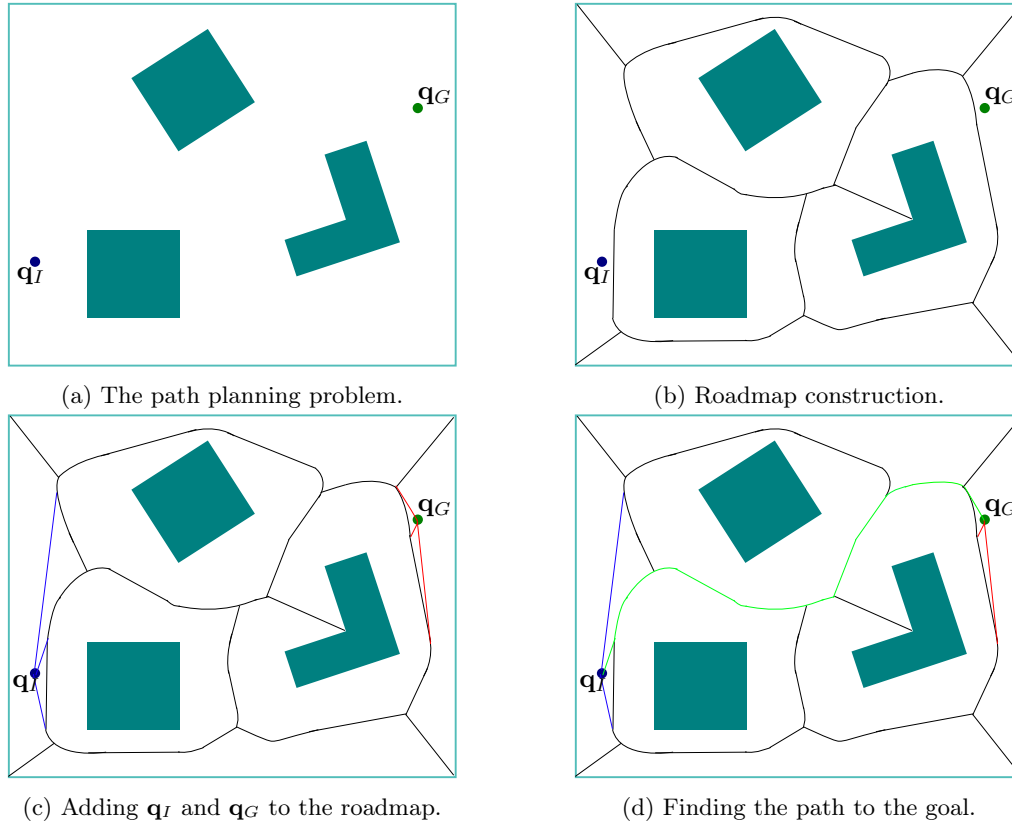


Figure 2.7: Path planning using Voronoi graphs.

Firstly, the decomposition must be done such that it is easy to compute a path between any pair of points that both lie inside a single cell. With the simplest path between two points in space being a straight line, the geometric property that each cell in the decomposition needs to possess so that any pair of points inside it can be connected with a straight line is that it must be convex. The second property pertains to the ease of traversing from one cell to another – this is useful for building the roadmap, and it requires the adjacency of cells to be well defined and easy to extract. The last desirable property has to do with the ease of connecting the initial and goal configurations to the roadmap. In particular, with the first two properties fulfilled, this property requires that it must be efficient to determine which cells of the decomposition contain q_I and q_G . We illustrate two cell decomposition strategies from literature, namely *vertical cell decomposition* and *triangulation* in Subsections 2.4.4.1 and 2.4.4.2 respectively.

2.4.4.1 Vertical Cell Decomposition

As its name suggests, vertical cell decomposition [42] proceeds by decomposing $\mathcal{C}_{\text{free}}$ using vertical lines and it is illustrated in Figure 2.8. In this case, the each resulting cell is either a trapezium, a triangle or rectangle or a square – the latter three being special versions of a trapezium. It is thus also referred to as *trapezoidal decomposition*. Most importantly, all cells are convex.

Once decomposition is done, an adjacency graph can be built, which stores information about the cells that can be accessed from each cell. Using the adjacency graph, the roadmap as shown in Figure 2.8(b) can then be built. The roadmap is constructed by first placing a point at the centroid of each cell as well as at the midpoint of a shared boundary as shown. The path connecting these points can then be drawn, one cell at a time, by connecting the each midpoint of a shared boundary to the centroid of the cell by simply using a straight line. Once the roadmap is constructed, any path planning-query in the domain of the planner can be answered. Given q_I and q_G , the first step in answering a path-planning query is to connect them to the roadmap via the centroids of the cells containing each of them – as shown in Figure 2.8(c). With this connection done, the planning

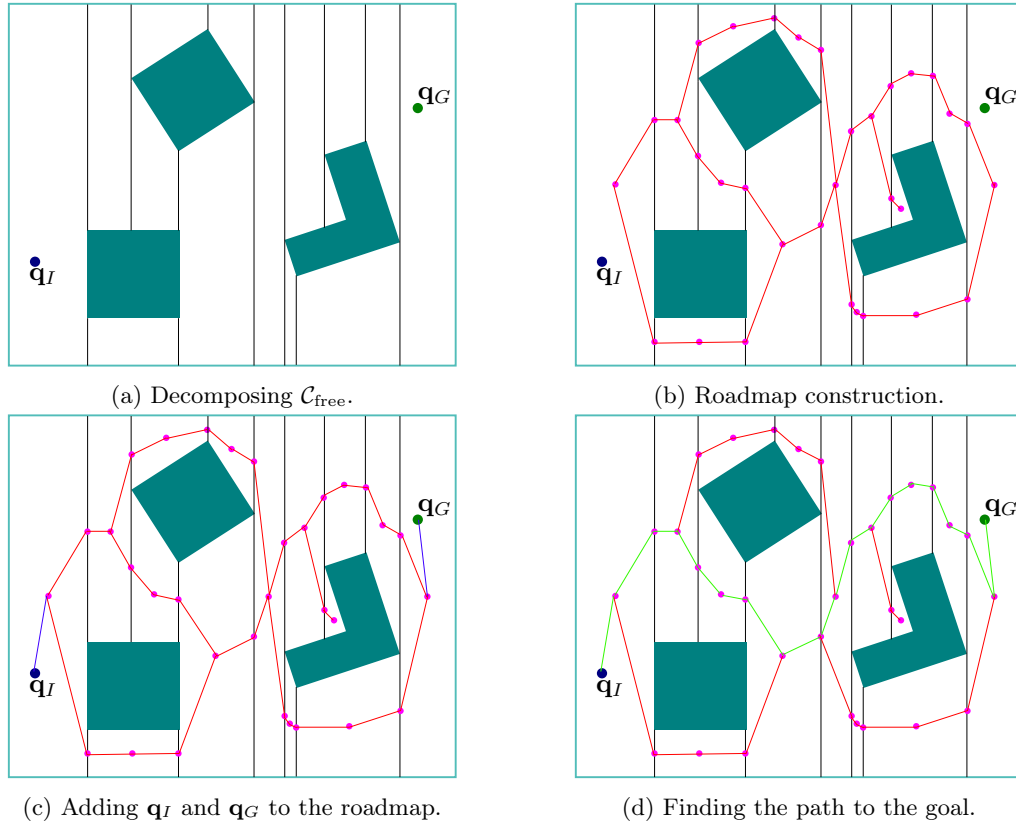


Figure 2.8: An illustration of the vertical cell decomposition method for path planning.

query is answered by performing a graph search as shown in Figure 2.8(d). We now move on to the second cell decomposition technique, triangulation.

2.4.4.2 Triangulation

Triangulation [25, p. 268] works the same way as the vertical cell decomposition technique of the preceding discussion, except that instead of breaking down the environment into trapezoidal cells, it is broken down into triangular cells as depicted in Figure 2.9(a). The construction of the roadmap (Figure 2.9(b)), addition of \mathbf{q}_I and \mathbf{q}_G to the roadmap (Figure 2.9(c)) and the searching of the roadmap for a solution path (Figure 2.9(d)) follows exactly the same procedure described for vertical cell decomposition.

The next subsection evaluates the cell decomposition techniques against the path planning requirements of this thesis.

2.4.4.3 Conclusion on Cell Decomposition

Cell decomposition method differs from the other types of combinatorial path planners in that it does not build the roadmap directly from obstacle boundaries. Instead, it first breaks the free space into small regions known as cells and then from these cells, builds the roadmap. There are many ways in which the decomposition can be done, but the rule of thumb is to employ a decomposition strategy that results in convex cells. The reason is that this makes it possible to simply connect any two points inside each cell using a straight line. The technique was illustrated using trapezoidal decomposition and triangulation. The resulting paths consist of concatenations of straight lines. Such paths have a discontinuous-curvature profile and are not suitable for vehicles with a constrained angular velocity and angular acceleration. As such, this technique is not suitable for our application. This is the last considered combinatorial path planner, the next subsection

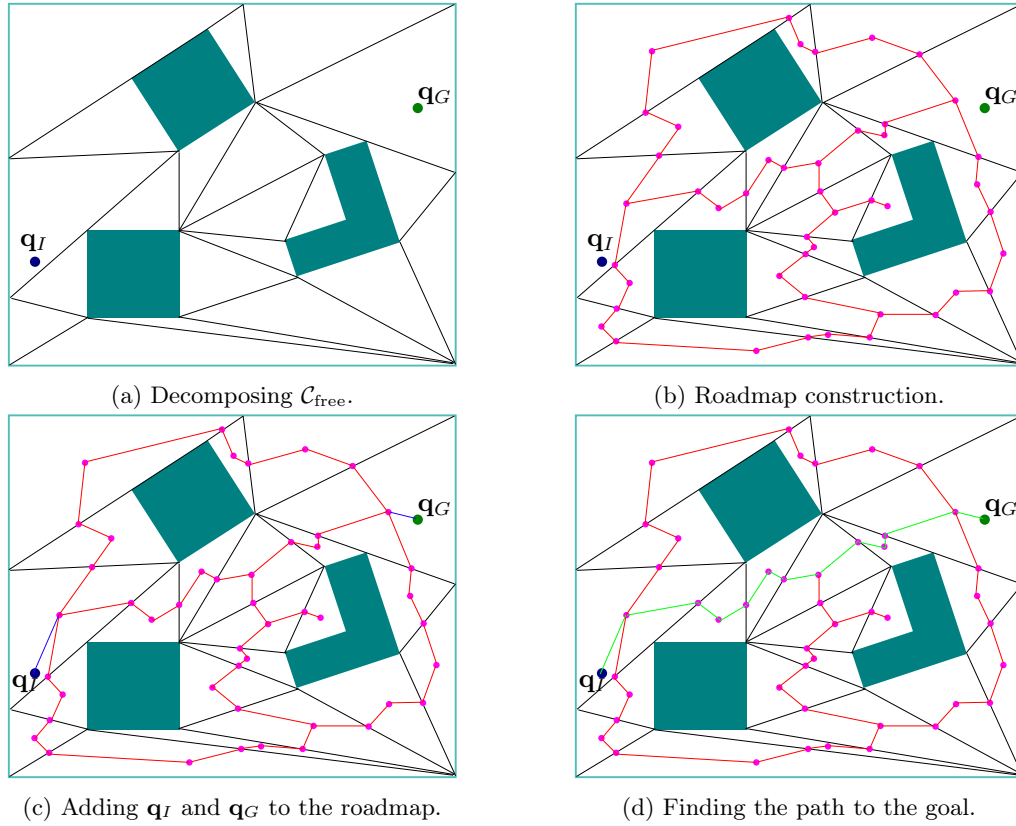


Figure 2.9: An illustration of the triangulation method for path planning using cell decomposition.

concludes our treatment of combinatorial path planning by summarising our findings on this path-planning approach.

2.4.5 Conclusion on Combinatorial Path-planning Techniques

Different techniques for combinatorial path planning have been considered. These are visibility graphs, Voronoi diagrams, and cell decomposition. They have different properties, in particular visibility graphs find the shortest path; however, they do so by grazing obstacle vertices – an undesirable attribute in mobile robotics since it increases chances of collisions. It was noted that to mitigate the risk of getting too close to obstacles, an obstacle clearance measure can be taken by inflating obstacles by a desired clearance amount prior to the application of the planner. In contrast to visibility graphs, Voronoi diagrams produce paths of maximum clearance from obstacles, but they do so at the expense of optimality in terms of path length. Cell decomposition techniques differed from the first two techniques in that they first break down the free space into cells before constructing the roadmap. If the decomposition strategy is inefficient, this may incur a significant computation time compared to the first two techniques that do not decompose $\mathcal{C}_{\text{free}}$.

The unifying properties of the different combinatorial path-planning techniques include their assumption of the exact knowledge of the shape and position of obstacles and that all these techniques do not produce paths that respect the dynamics of the vehicle. In particular, contrary to the path planning requirements of this project, the paths produced are not continuous in curvature. As a result, our position is that using these techniques to plan paths that will be executed by a vehicle with a constrained turning rate and angular acceleration defeats the purpose of path planning since the vehicle cannot (even in the absence of disturbances) follow this path exactly. As such, the vehicle would be exposed to chances of collision. It is for these reasons that these path-planning techniques have not been selected for use in this project. The search for suitable path-planning methods continues in the next two sections, with grid-based path planning studied next.

2.5 Grid-based Path Planning

Grid-based (also known as *discrete*) path planning is widely thought of as the most straightforward approach to path planning [43]. It differs from the combinatorial path-planning approach discussed in the preceding section in the manner in which it characterises \mathcal{C}_{obs} and how it searches $\mathcal{C}_{\text{free}}$. Unlike the combinatorial approach, it does not exactly model \mathcal{C}_{obs} , but resorts to approximating it. With the configuration space concept used to model the robot's operating environment, the grid-based path-planning approach quantises the configuration space by imposing a regular grid of a specified resolution on it [44, 45]. It then iterates over all cells of the grid using a classifier that assigns some cost to each cell. Two representations for the cost exist, namely *occupancy grids* and *grid costmaps*. In occupancy grids, a cell is either free or occupied and in terms of costs, free cells are assumed to have a low cost while occupied cells are assigned very high costs. Occupancy grids are typically used in indoor environments where most detected obstacles are rigid and attempting to drive through any of them can endanger the robot. This is unlike the case of outdoor environments where there are some obstacles such as tiny rocks, bumps and grass, over which the robot can drive – for outdoor environments grid costmaps are more relevant. In the case of occupancy grid representation, the classifier used to assign costs to cells can be viewed as a *collision detector* which marks each cell as either occupied or unoccupied [43]. This results in the rasterisation of configuration space obstacles into a group of grid cells [46] as illustrated in Figure 2.10. Figure 2.10(a) shows an example configuration space and Figure 2.10(b) shows an example discretisation of this configuration space for grid-based path planning. While the illustration of Figure 2.10(b) classifies each cell as either occupied or not, another approach is to make use of a *probabilistic collision detector* which instead assigns a probability of occupancy to each cell. Grid costmaps are slightly different from occupancy grids. They differ in that while occupancy grids encode information about the likelihood of an obstacle being present in each cell, grid costmaps encode the traversability of the part of the environment represented by each cell. In creating costmaps, the environment is classified into different terrain types, such as grass, tar, dirt road, grass, trees and based on which terrain types a particular cell consists, a proportional cost is assigned. The cost is assumed to range from “free” to “lethal”. The resulting representation is known as a *scalar costmap*, i.e. the cost assigned to each cell is a scalar value. In the probabilistic case, the representation is known as a *probabilistic costmap* and the cost assigned to each cell is a probability distribution that captures all possible terrain types for that cell and their associated probabilities. With the grid costmap created, the path planning problem then reduces to searching this costmap for the minimum cost path that leads from the initial configuration, \mathbf{q}_I to the goal configuration, \mathbf{q}_G .

To facilitate this search, the grid costmap is converted to a graph whose nodes are either centres or corners of grid cells. For each node, an adjacency relationship specifying how neighbouring nodes can be reached is defined. A common adjacency relationship is one in which it is possible to move one step “up”, “down”, “left” or “right” from the current node. It is also possible to describe the adjacency relationship in angular terms. In this case, the preceding example can be represented as 90° increments. If diagonal movements from the current node are allowed, 45° increments can be used. At the extreme are techniques in which any angle is permissible [47] and in which transition is not restricted to neighbouring nodes, but is allowed for all graph nodes. There is no right or wrong choice for this adjacency relationship; it depends on the nature of the application and the onus of making a suitable choice is left to the system designer.

With the grid costmap converted to a planning graph and the adjacency relationship defined, a path between any two configurations (\mathbf{q}_I and \mathbf{q}_G) can be found by applying a search algorithm on the graph. Depending on how the costs of cells are represented, there are different approaches to grid-based path planning. For the case where the cost assigned to each cell is a scalar value (scalar costmaps), classical AI graph search techniques such as dynamic programming [48], A* search [49] or Dijkstra's algorithm [41] can be directly applied to the costmap to find a path to the goal. However for the case where there is uncertainty in the costs of cells (whether due to the cell containing a mixture of terrain types or where probability of occupancy is specified for each cell), there is a need to handle this uncertainty. Two paradigms exist for this purpose, namely *assumptive planning* [50] and *planning under uncertainty* (also known as *decision-theoretic planning* [25]). Assumptive planning proceeds by making a simplifying assumption about the costs of cells. In the

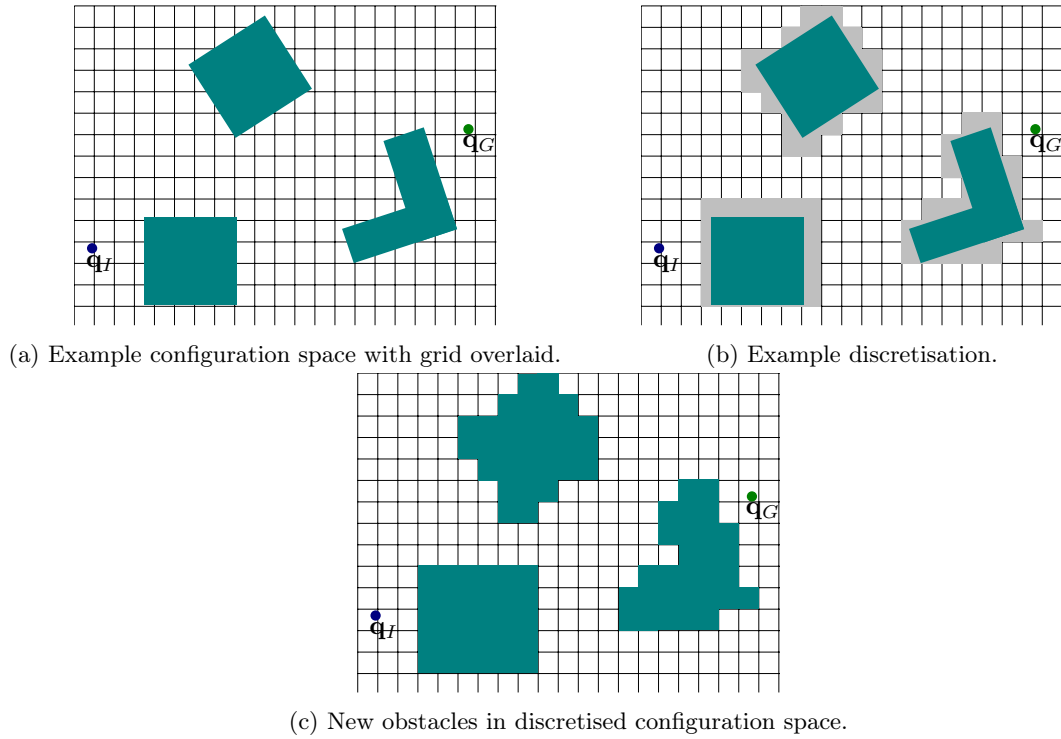


Figure 2.10: An illustration of configuration space discretisation for grid-based path planning.

case of probabilistic occupancy, cells that are most probably occupied (based on a set probability of occupancy threshold) are classified as occupied. Similarly, in the case of probabilistic costmaps, a cell which is most probably a dirt road, although it might be a degraded lawn, is classified as a dirt road and assigned the cost of traversing a dirt road. With this simplifying assumption made, planning can then proceed as in the case of scalar costmaps. On the other hand, decision-theoretic planning handles uncertainty directly and in a principled manner, without making such simplifying assumptions. In both cases, due to the discretisation of the configuration space, the resulting path planners have a weaker notion of completeness called *resolution completeness*. This means that the path planner can be guaranteed to find a path from any initial configuration to any goal configuration (if one exists) if the grid resolution is fine enough. Such path planners plan paths quickly on grids with low grid resolutions and they are prohibitively slow on finer grids. Furthermore, just like the combinatorial approach, this approach suffers from the curse of dimensionality as the number of grid cells grows exponentially with an increase in the dimension of the configuration space. In the next two subsections, we review the two approaches to grid-based path planning, namely assumptive planning (which we also refer to as planning in scalar costmaps) and grid-based path planning under uncertainty.

2.5.1 Assumptive Path-planning in Grid Costmaps

In the absence of uncertainty in the costs assigned to cells, grid-based path planning proceeds by performing a graph search to find a sequence of grid cells that form a path from the initial configuration to the goal configuration. In the case of uncertainty in cell costs, one way to handle such uncertainty is through assumptive planning – making simplifying assumptions about cell costs as described in the introduction of this section. This reduces the problem to that of *planning in scalar costmaps*. In this subsection, we review existing algorithms for searching planning in such costmaps. While the planning techniques of this subsection were not selected as suitable for the path-planning requirements of this project, they share a lot of similarities with the sampling-based path planners of Section 2.6, which were found to be suitable. In fact ideas behind some assumptive grid-based path planners have inspired their sampling-based counterparts over the years. For this

reason, it is worthwhile to study assumptive grid-based path-planning techniques to a greater level of detail. We now introduce the fundamentals of assumptive grid-based path planning.

2.5.1.1 Fundamentals of Grid-based Path-planning using Graph Search

Planning in scalar costmaps is handled using graph search; however, the graph being searched is not known beforehand, but it is revealed incrementally as part of the planning process by attempting to visit cells in the grid [25]. There are two ways to construct this graph. *Explicit* graph representations build the entire graph before beginning to plan, while *implicit* graph representations build the graph on-the-fly as the planning progresses, attempting to visit neighbouring cells only when necessary. We denote the planning graph by $\mathcal{G} = (V, E)$. It has a vertex set, V and an edge set E . Its vertices correspond to grid cells and the edges correspond to transitions between these grid cells. Vertices corresponding to cells containing \mathbf{q}_I and \mathbf{q}_G are respectively denoted by v_I and v_G . The path-planning problem is to find a path from v_I to v_G . If it is desired for such a path to be the shortest possible path from v_I to v_G , then the problem is known as the *single-source shortest path problem* (also known as the point-to-point shortest path problem [51]). To ensure completeness in planning by graph search, the search algorithm has to be *systematic* [25]. In essence, for a finite graph, being systematic requires the search algorithm to guard against visiting already visited graph vertices so as to avoid redundant exploration which may cause it to run forever. A systematic search algorithm categorises vertices according to some criteria such as *unvisited*, *dead*, and *alive* as the search progresses. The categories *open* and *closed* are also common. Under the first categorisation, unvisited vertices are those which have not been visited yet. At the beginning of a search, all vertices are unvisited. Dead vertices are those that have been visited and whose neighbours have all been visited too – they offer no new information in the search. Alive vertices are those that have been encountered, but not all their neighbours have been visited. The second categorisation is more concise – open vertices are those which are not yet fully examined (i.e. both unvisited and alive) and closed vertices are those that have been fully examined (i.e. dead). Different grid-based path-planning algorithms differ in the manner in which open vertices are scheduled for visitation and how they examine a visited vertex. This depends on the type of plan that the algorithm aims to produce. It may be that the algorithm is tailored to find the optimal plan (optimal path planning) or that it has to generate an initial plan and keep updating it to account for moving obstacles (incremental replanning), or that it has to find an initial path quickly and then spend the rest of the planning time improving it (anytime path planning) or that it has to find the initial path quickly and then spend the remaining time improving the solution and repairing it to account for obstacle movements (anytime replanning). In the rest of this subsection, we discuss four categories of grid costmap path planners, namely *optimal path planners* (Section 2.5.1.2), *incremental replanners* (Section 2.5.1.3), *anytime path planners* (Section 2.5.1.4) and *anytime replanners* (Section 2.5.1.5).

2.5.1.2 Optimal Path-planning in Grid Costmaps

Path planning is the task of computing a path that takes a robot from its initial configuration, \mathbf{q}_I , to a goal configuration, \mathbf{q}_G . Optimal path planning is a specific path-planning problem in which the resulting path has to be one with the least cost. With path length as the objective function, the problem is that of finding the shortest path from \mathbf{q}_I to \mathbf{q}_G . This subsection discusses algorithms used for computing optimal paths in grid costmaps using graph search. We begin our discussion with Dijkstra's algorithm which is discussed next and conclude with the A* algorithm, which is an improvement to Dijkstra's algorithm.

2.5.1.2.1 Dijkstra's algorithm

Dijkstra's algorithm [41], introduced by Edsger Wybe Dijkstra in 1959, addresses the problem of finding the shortest path in a graph, $\mathcal{G} = (V, E)$, with a vertex set, V , and edge set, E – where all edge costs are strictly positive. Again, the vertices corresponding to cells containing \mathbf{q}_I and \mathbf{q}_G are respectively denoted by v_I and v_G . Dijkstra's algorithm addresses this problem by finding the shortest path from v_I to every other vertex on the graph. The cumulative cost from v_I to a given

vertex on the graph is known as the *cost-to-come* or the *g-value* of that vertex and is denoted by $g(v)$.

Algorithm 1: Dijkstra's algorithm

Input: Graph $\mathcal{G} = (V, E)$, Initial vertex v_I , Goal vertex v_G
Output: A path from v_I to v_G

```

1 foreach  $v \in V$  do
2   if  $v \neq v_I$  then
3      $g(v) \leftarrow \infty$  ; // unvisited vertices' cost-to-come unknown, set to a large value.
4  $g(v_I) \leftarrow 0$  ; // cost-to-come for initial vertex.
5  $\text{CLOSED} \leftarrow \emptyset$  ; // a list that stores visited vertices – no vertex visited yet.
6  $\text{OPEN} \leftarrow V$  ; // priority queue to store vertices that are yet to be visited.
7 while  $\text{OPEN} \neq \emptyset$  do
8    $v' \leftarrow \text{OPEN.getFirst}()$  ; // pop-out vertex with least cost-to-come from the queue.
9    $\text{CLOSED} \leftarrow \text{CLOSED} \cup \{v'\}$  ; // mark the visited vertex as closed.
10  if  $v' == v_G$  then
11    return  $\text{constructPath}(v_G)$  ; // recover path by tracing parent vertices.
12  foreach  $v \in \text{successors}(v')$  do
13    if  $g(v') + c(v', v) < g(v)$  then
14       $g(v) \leftarrow g(v') + c(v', v)$  ; // update cost-to-come.
15       $\text{parent}(v) \leftarrow v'$  ; // update parent.
```

Pseudocode for the algorithm is shown in Algorithm 1. The algorithm makes use of two lists containing the graph vertices. The set CLOSED (line 5) stores vertices whose shortest path from v_I has been determined. Initially, with no vertex visited by the algorithm, CLOSED is empty. The set, OPEN , (line 6) is a priority queue, storing vertices which are yet to be visited by the algorithm. Initially, OPEN contains all the vertices of the graph. OPEN is sorted according to the *g-value*, which is the best known cost-to-come so far for each vertex, $v \in \text{OPEN}$. Initially, the *g-value* is initialised to 0 for the initial vertex while for all other unvisited vertices, it is initialised to ∞ (lines 1–4). In each iteration (of the while loop), the algorithm selects an unvisited vertex v' in OPEN with the lowest *g-value* and removes it from OPEN (line 8) – initially, v_I has the least cost and it is the one selected. With the vertex with the lowest cost selected, it is added to the set of visited vertices, CLOSED (line 9). The algorithm then checks if reaching each neighbour (also known as a successor) v of v' via v' improves v 's *g-value*. If this is the case, the *g-value* of v is updated (line 14) and its parent vertex is also updated (line 15). Neighbours to a given vertex are selected by the function $\text{successors}(v)$ (line 12). In its most common formulation, the algorithm will eventually visit all graph vertices until the priority queue, OPEN , becomes empty and CLOSED contains all graph vertices. However, a modification (line 10–11) can be done to halt the algorithm immediately once v_G has been visited. At this point, the cost of the shortest path from v_I to v_G is simply the v_G 's *g-value*, i.e. $g(v_G)$. The shortest path from v_I to v_G can be recovered by back-stepping through parent vertices, starting from v_G , until v_I is reached.

While the algorithm is guaranteed to return the shortest path from v_I to v_G , it finds this path by visiting all vertices on the graph, finding the shortest path from v_I to every vertex on the graph – hence it is computationally inefficient. The next subsection introduces a more efficient algorithm that reduces the number of vertices visited in the pursuit of the optimal path in a grid costmap.

2.5.1.2.2 A* Search

Dijkstra's algorithm described in the previous subsection does well in finding the shortest path from v_I to v_G , but it does so by visiting all graph vertices and computing each vertex's shortest path from v_I . This makes the algorithm slow. The A* algorithm [49] is an improved algorithm that reduces number of vertices visited in the quest to find the shortest path from v_I to v_G . The idea is to introduce a *cost-to-go* heuristic, also known as the *h-value* and denoted by $h(v)$ in addition to the *cost-to-come* metric, denoted by $g(v)$, used by Dijkstra's algorithm. As a result, in

A*, the priority queue, OPEN, is not only sorted based on the cost of vertices from v_I , but also according to an estimated cost to reach the goal vertex, v_G . Equation 2.1 gives the estimated cost of reaching v_G via a vertex v , the component, $g(v)$, represents the cost-to-come and $h(v)$ is the heuristic cost-to-go. This estimated cost is also known as the f -value, which is defined as

$$f(v) = g(v) + h(v). \quad (2.1)$$

This results in an algorithm which biases its search towards the goal and ignores vertices which in spite of having a low cost-to-come, have a high overall estimated cost (f -value) to reach v_G . Owing to this behaviour, the algorithm is sometimes referred to as the *heuristic search* or the *goal-directed search* [51]. The reason the cost-to-go is a heuristic and not exact is that it is not possible to know the exact cost-to-go beforehand [25]. A required property of the cost-to-go heuristic is that it should never overestimate the true cost-to-go (i.e. it should be always less than or equal to the true cost-to-go). If this property is ensured, the goal bias does not impair the algorithm from finding the optimal path. As the heuristic approaches the value of the true cost-to-go, the algorithm considers much fewer vertices than Dijkstra's algorithm – hence more efficient. In grid-based path planning, reasonable underestimates of the cost-to-go include the Euclidean or Manhattan distance from the vertex under consideration to v_G – these are guaranteed to always underestimate the true cost-to-go as the presence of obstacles can only increase the path length. The algorithm's pseudocode is shown in Algorithm 2.

Algorithm 2: A* search

Input: Graph $\mathcal{G} = (V, E)$, Initial vertex v_I , Goal vertex v_G
Output: A path from v_I to v_G

```

1 foreach  $v \in V$  do
2   if  $v \neq v_I$  then
3      $g(v) \leftarrow \infty$  ; // unvisited vertices' cost-to-come unknown, set to a large value.
4  $g(v_I) \leftarrow 0$  ; // cost-to-come for initial vertex.
5  $\text{CLOSED} \leftarrow \emptyset$  ; // a list that stores visited vertices – no vertex visited yet.
6  $\text{OPEN} \leftarrow \emptyset$  ; // priority queue to store vertices that are yet to be visited.
7  $h(v_I) \leftarrow h(v_I, v_G)$  ; // estimated cost-to-go for the initial vertex.
8  $\text{OPEN} \leftarrow v_I$  ; // insert initial vertex (with cost:  $g(v_I) + h(v_I)$ ) into OPEN list.
9 while  $\text{argmin}_{v \in \text{OPEN}} (g(v) + h(v, v_G)) \neq v_G$  do
10    $v' \leftarrow \text{OPEN.getFirst}()$  ; // pop out vertex with least cost-to-come from queue
11    $\text{CLOSED} \leftarrow \text{CLOSED} \cup \{v'\}$  ; // mark the visited vertex as closed.
12   if  $v' == v_G$  then
13     return  $\text{constructPath}(v_G)$  ; // recover path by tracing parent vertices.
14   foreach  $v \in \text{successors}(v')$  do
15     if  $g(v') + c(v', v) < g(v)$  then
16        $g(v) \leftarrow g(v') + c(v', v)$  ; // update cost-to-come.
17        $\text{parent}(v) \leftarrow v'$  ; // update parent.
18        $h(v') \leftarrow h(v', v_G)$  ; // estimated cost-to-go for  $v'$ .
19        $\text{OPEN} \leftarrow \text{OPEN} \cup \{v'\}$  ; // insert  $v'$  (with cost:  $g(v') + h(v')$ ) into OPEN.
```

The pseudocode has been adapted from the work of Fegurson [52], but re-written in consistency with Dijkstra's algorithm presented in Algorithm 1 for ease of comparison. As shown by the pseudocode, A* resembles Dijkstra's algorithm in some respects, but has a number of differences. Lines 1–5 are exactly the same – these lines correspond to the initialisation of the cost-to-come metric for all vertices on the graph and that of the CLOSED list. Line 6, which initialises the priority queue in both algorithms differs from that of Dijkstra's algorithm, where the priority queue was initialised to contain all graph vertices. This is because Dijkstra's algorithm aims to eventually visit all vertices on the graph. However, for the A*, this is not the case – particularly, it is not possible to know beforehand which vertices will be visited by the algorithm as this is determined online based on the f -values of encountered vertices as the algorithm runs. For this reason, with no vertex encountered, the priority queue is initially empty. Once the initial vertex has been visited

and its heuristic cost-to-go determined (line 7), it is found to have the least f -value than all vertices on the graph, hence it is added to OPEN (line 8). Each iteration of the algorithm (line 9–19), begins by popping out the vertex, v' , with the least f -value from OPEN (line 10). The popped vertex is then marked as visited by being added to the CLOSED list (line 11). If this vertex is v_G , then the solution has been found (line 13), otherwise the algorithm updates the cost of all its neighbours (vertices reachable through a direct edge from v') as follows: if the sum of the g -value of v' and the cost of the edge from v' to a neighbouring vertex, v , ($c(v', v)$) is less than the v 's current g -value, then v 's g -value is updated to this new, lower value (line 16). The parent vertex of v is also updated to v' (line 17) and the h -value is recomputed and updated (line 18). Such a neighbour whose cost has reduced is then inserted in the OPEN list which is sorted according to f -values. The while loop iterates until the goal vertex, v_G , is popped off from OPEN (line 12), at which point, the solution is found and the path can be recovered by back-stepping through parent vertices, starting from v_G , until v_I is reached.

The A* algorithm described above is efficient in planning the shortest path through a planning graph that is fully known beforehand [52], i.e. where the configuration space is fully known at the start of the planning process and does not change thereafter. This is not an issue in this project, since planning is done in static environments. Algorithms, known as replanners, that are capable of planning in environments that are not fully known at the start of planning, or may change while planning is underway are discussed in the next subsection. They are outside the scope of this project since planning is done in static environments. Therefore, they are only considered for completeness sake.

2.5.1.3 Incremental Replanning in Grid Costmaps

When planning in dynamic environments, it becomes necessary to keep track of detected changes in the environment and to produce an updated plan, in case portions of the current plan are invalidated by the movement of obstacles. This is known as *replanning* and since it is outside the scope of the project, we only present a high-level overview of existing replanning algorithms, for the sake of completeness.

There are two ways by which replanning can be achieved. One option is to start planning afresh, for example by running a new A* search, and the other is to repair the invalidated parts of the plan. Among these options, replanning from scratch may be computationally inefficient since it may happen that only small portions of the original plan are invalidated. Also, replanning may be required frequently, depending on how quickly the environment changes or how much of it is unknown prior to the start of planning. Therefore, replanning from scratch in these instances may be computationally expensive. The D* (short for dynamic A*) search [53–55] and algorithms based on it [56, 57] are capable of replanning by repairing invalidated portions of planned paths. An important capability that these algorithms rely on is the ability to reverse A*'s search direction, i.e. performing planning from the goal vertex and towards the initial vertex. This behaviour is known as the *backwards* A*. The D* algorithm is a successor to other algorithms [58, 59] that were introduced before it to handle replanning by repairing invalidated portions of an initial plan. These predecessors of D*, however, handled replanning by locally modifying the planned path so as to avoid newly detected obstacles. As a result, while these approaches were able to re-plan, they were suboptimal since the local modifications were done without updating the path globally to ensure that it has the lowest cost possible after these changes. D* and its variants repair invalidated paths while also retaining global optimality. Notable variants of D* are focused D* [56] and D* lite [57].

The D* algorithm's invention is a result of a number of observations about how a robot moves through its environment as it executes a planned path and the nature of the new information it receives while executing this plan. In particular, the first observation is that the robot typically makes use of a range sensor that has a short range. This implies that changes can only be detected within this short range around the robot. As such, on the planning graph, edge costs that will change are those that are in the vicinity of the robot's current location. This implies that the planned path will need to be repaired locally and the changes propagated to the rest of the plan to retain global optimality. The second observation is that since changes in the environment are observed as the robot moves towards v_G and away from v_I , the part of the initial path that needs to be repaired reduces progressively as the robot approaches the goal. The algorithm resembles

A* in that it also uses heuristics to limit vertices visited in the planning process. It differs from A* on where the planning starts and ends. Unlike in A* where the search starts from the initial vertex, v_I , and proceeds towards the goal vertex, v_G , in D*, the search starts from v_G and ends at the robot's current location – i.e. it is a backwards A*. Consequently, the g -value is zero at the goal vertex in a D* search. From a replanning perspective, given the fact that changes in edge costs on the graph as detected by the robot's range sensor occur close to the robot's current position, by planning backwards from v_G and towards the robot's current position, the number of child vertices affected by this change and for which this change in costs needs to be propagated is smaller as changes occur in the vicinity of the robot and changes are propagated towards the robot's current position. This is unlike the case of planning in the forward direction from the robot's current position towards v_G , where the number of child vertices for which changes in edge costs would have to be propagated is significantly larger as v_G is far away from where changes occur as compared to the robot's current position.

There are a number of improvements to the D* algorithm that have been presented in literature since its inception. In the rest of this discussion, we step through some notable evolutions of the algorithm since its inception, starting with the original version. The first version of the algorithm [53–55] is functionally equivalent to replanning from scratch using A*, but it is far more efficient [53]. Just like A*, the algorithm [54] maintains an OPEN list which contains vertices which are yet to be evaluated. The difference is that in D*, the OPEN list is also used to keep track of vertices whose costs have changed and to propagate this change to affected child vertices. The cost assigned to each vertex is a heuristic estimate from the vertex to v_G , i.e. $h(v_G, v)$. Unlike A*, the path cost used by D* does not include a term that measures the cost of a vertex from the robot's current position. Consequently, when a vertex affected by edge cost changes is processed, cost changes are propagated in all directions from the vertex, without any bias of such cost propagations towards the robot's position. This makes it similar to a backwards A* search with $h(v) = 0$ or a backwards Dijkstra search. An improvement to D* came forth in the form of the *focused D** algorithm [56] which made use of a heuristic that focuses the propagation of the costs of vertices whose costs have changed towards the robot's current position. This focusing heuristic $g(v, R_i)$ estimates the cost of the path from the robot's current location, denoted by R_i , to a vertex, v . This results in an f -value given by Equation 2.2. Similarly to the f -value of A*, this f -value,

$$f(v, R_i) = h(v_G, v) + g(v, R_i), \quad (2.2)$$

gives an estimated cost from the robot's current location via vertex, v , to the goal vertex. Since focused D*'s f -value depends on the robot's current location, R_i , which changes as the robot continues to execute the plan, depending on when a vertex was added to OPEN, the value of R_i used in its f -value computation differs. However, since the vertices in the OPEN list priority need to be sorted using the f -value in a fair manner, it is necessary to compensate for this dependence of the f -value on when the vertex was inserted to the queue. For this purpose, the modified f -value used to sort the queue includes a compensation term, $d(R_i, v_I)$, which is the distance from the initial vertex to the robot's current position. The compensated f -value formula, given by

$$f(v) = f(v, R_i) + d(R_i, v_I), \quad (2.3)$$

allows for vertices to be sorted fairly, according to the estimated cost of the path from the initial vertex via each of them and to the goal vertex. Using this compensated f -value, focused D* focuses the propagation of costs towards the robot's current position, thereby finding new optimal solutions by visiting fewer vertices compared to D*. However, both the D* and focused D* algorithms are considered difficult to understand, implement and extend [52, 62]. An algorithm that is fundamentally very similar to focused D* [62], yet algorithmically simple to understand, implement and extend, and more efficient [52, 62] is D* lite [57]. The simplicity and efficiency of D* lite has been attested by the developers of D* and focused D*, who chose to extend D* lite instead of either of their two algorithms, citing this simplicity and efficiency [60, 63].

Although it has not been explicitly stated up to now, transitions between cells in D*, focused D* and D* lite are restricted to moving from the centre of a grid cell to the centre of any of its neighbouring cells. This results in paths that are jagged. An algorithm that improves this aspect

of these planners is field D* [61]. This algorithm tries to reduce the jerkiness of the paths by allowing transitions to any point on the edge of a grid cell.

We have now reviewed two of the four categories of grid costmap path planners. The reviewed categories are optimal path planners and incremental replanners. The next subsection reviews anytime path planners, with anytime replanners being the subject of Subsection 2.5.1.5.

2.5.1.4 Anytime Path-planning in Grid Costmaps

In the previous subsection, we reviewed grid-based path-planning techniques that are able to plan an initial optimal path and to repair the planned path without replanning from scratch if changes in the environment invalidate parts of the initial plan. We now consider algorithms suited for situations in which the robot is required to react quickly, yet the path-planning problem is complex. In such situations, planning optimal paths as those computed by the path planners of the previous two subsections can be impossible due to the time needed to generate such optimal plans. With time constraints on the planning problem, we must be content with the best path that we can generate within the confines of the available time. Path-planning algorithms that address this problem are known as *anytime path-planning algorithms*. Such algorithms [13, 14] typically work by constructing an initial, possibly highly suboptimal, path quickly and then continually improve this path as time permits. Once the initial path has been generated, the robot can start executing it, and at the same time the improvement of the solution can continue on the remaining part of the path.

Planning in anytime fashion requires that we formulate the grid-based path planner so that it finds an initial path quickly. A way of achieving this with the A* algorithm of Subsection 2.5.1.2.2 as a baseline is to modify the algorithm by inflating its heuristics [64]. This inflation amounts to multiplying the actual heuristic values by an inflation factor (also known as a sub-optimality bound) $\epsilon > 1$ [15]. The term *sub-optimality bound* refers to an upper limit on the costs of paths that the anytime path planner may find. In particular, for A* with inflated heuristics, the inflation factor, ϵ , is the sub-optimality bound. The algorithm is guaranteed to find a path that costs that is at most ϵ times the cost of the true optimal solution, otherwise less. With $\epsilon = 1$, the algorithm reduces to A* and it is guaranteed to exactly find the optimal path. A naive way of implementing anytime A* with sub-optimality bounds would be to choose a set of inflation factors and then run successive A* searches with each inflation factor in descending order. While this approach can produce a series of solutions, with each new solution having a lower sub-optimality bound, it is computationally inefficient since each inflated A* run starts planning from scratch and essentially duplicates some of the computations performed by earlier runs. For this reason, it is worthwhile to reuse computations performed by earlier runs as much as possible in each successive run. In the rest of this discussion on grid-based anytime path planning, we start by discussing A* with inflated heuristics before discussing some notable adaptations of A* for anytime path planning.

2.5.1.4.1 A* with an Inflated Heuristic

In improving the efficiency of searching the state space, A* improves Dijkstra's algorithm by introducing a heuristic, $h(v)$, which reduces the number of vertices visited during the search. As stated in the discussion on D* lite (Subsection 2.5.1.2.2), for this heuristic to be considered admissible, it must never overestimate the actual cost of reaching the goal from the vertex under consideration. Inflating the heuristic, i.e. using $h'(v) = \epsilon * h(v)$ instead of $h(v)$, often results in much fewer vertex expansions and therefore faster searches. However, an undesirable effect of inflating the heuristic is that doing so may violate the admissibility of the heuristic. As a result, the resulting planner loses the A*'s guarantee to find the optimal path. This planner is A* with an inflated heuristic. It is also called *weighed A**. In situations where path planning has to be done under time constraints, we may be content with settling for the best suboptimal solution that can be found within the allocated time. As such, although inflating the heuristic removes the guarantee for optimality, it is useful in anytime path planning. The only difference between weighed A* and A*, whose pseudocode was given in Algorithm 2, is the multiplication of the heuristic (h -value) in lines 7 and 18 by the inflation factor, ϵ . Since this change is minor and for the sake of brevity, we do not include pseudocode for the weighed A* algorithm. We now discuss algorithms that make use of weighed A* for anytime path planning in grid costmaps.

2.5.1.4.2 Anytime A*

The anytime A* (ATA*) algorithm [64] developed by Zhou and Hansen in 2002 was the first to implement an anytime heuristic search algorithm [15]. They built on an idea initially suggested by Ikeda and Imani [65] to reduce the size of the OPEN list priority queue by pruning vertices whose f -value is equal or greater than an upper bound previously established by a weighed A* search. Their algorithm also uses weighed A* to find an initial solution, which might be suboptimal, quickly. This initial solution provides an upper bound which is used to prune the priority queue. The weighed A* search is then continued with the aim of finding better solutions, which are also used as upper bounds for pruning the priority queue for each subsequent search. Pseudocode for the algorithm is given in Algorithm 3. The algorithm starts by setting the g -value of the initial vertex to zero (line 1). In line 2, the f -value of the initial vertex is computed as a sum of the g -value and the inflated version of the h -value, i.e. $\epsilon * h(v_I)$. The initial vertex is then inserted to the OPEN list priority queue (line 3) and the CLOSED list is initialised to an empty set since no vertex has been visited yet. Then at each iteration, the algorithm begins by popping off the vertex with the least cost from the OPEN list (lines 6–7), and marking it as visited by adding it to the CLOSED list (line 8). The popped vertex is then examined. If it is the goal vertex, then a new suboptimal solution has been found. The cost of this solution is computed in line 10 and it represents a bound that can be used to select vertices in OPEN that cannot possibly lead to better paths for pruning. This pruning is performed in lines 12–14. The rest of the while loop examines the neighbours of the visited vertex, updating their g -values and inflated h -values. Each neighbour whose f -value with the visited vertex as a parent would be less than the sub-optimality bound is examined. The f -value for such a vertex is updated if it is neither in the OPEN list nor in the CLOSED list or if the prospective g -value (i.e. its g -value with the visited vertex as a parent) is cheaper than its current g -value (lines 16–18). Each vertex whose g -value is updated is then inserted in the OPEN list and removed from the CLOSED list if it was previously inserted into CLOSED.

Algorithm 3: Anytime A*

Input: Graph $\mathcal{G} = (V, E)$, Initial vertex v_I , Goal vertex v_G
Output: A path from v_I to v_G

```

1  $g(v_I) \leftarrow 0$  ; // g-value for the initial vertex
2  $f(v_I) \leftarrow g(v_I) + \epsilon * h(v_I)$  ; // f-value for the initial vertex
3  $\text{OPEN} \leftarrow \{v_I\}$  ; // Insertion of initial vertex to OPEN list
4  $\text{CLOSED} \leftarrow \emptyset$  ; // CLOSED list initialised to an empty set
5 while  $\text{OPEN} \neq \emptyset$  do
6    $v' \leftarrow \text{OPEN.getFirst}()$  ; // Pop out the vertex that is at the front of OPEN
7    $\text{OPEN} \leftarrow \text{OPEN} \setminus \{v'\}$ ;
8    $\text{CLOSED} \leftarrow \text{CLOSED} \cup \{v'\}$  ; // Mark popped out vertex as visited
9   if  $v' == v_G$  then
10     $\text{bound} \leftarrow g(v') + h(v')$  ; // New cost bound
11    Output solution path and bound;
12    foreach  $v \in \text{OPEN}$  do
13      if  $g(v) + h(v) > \text{bound}$  then
14         $\text{OPEN} \leftarrow \text{OPEN} \setminus \{v\}$  ; /* Remove vertices that cannot lead to a better solution
           from OPEN */
15    foreach  $v_i \in \{v \mid v \in \text{successors}(v'), g(v') + c(v', v) + h(v) < \text{bound}\}$  do
16      if  $v_i \notin \text{OPEN} \cup \text{CLOSED}$  or  $g(v_i) > g(v') + c(v', v_i)$  then
17         $g(v_i) \leftarrow g(v') + c(v', v_i)$  ; // Update g-value of vertex
18         $f(v_i) \leftarrow g(v_i) + \epsilon * h(v_i)$ ; // Update f-value of vertex
19         $\text{OPEN} \leftarrow \text{OPEN} \cup \{v_i\}$  ; // Insert vertex into OPEN list
20        if  $v_i \in \text{CLOSED}$  then
21           $\text{CLOSED} \leftarrow \text{CLOSED} \setminus \{v_i\}$  ; // Remove vertex from closed list
  
```

While the anytime A* algorithm described above introduced interesting ideas that enable anytime planning in grid costmaps, it has two issues [15]. Firstly, the algorithm does not guarantee to visit each vertex at most once. The guarantee to visit each vertex at most once is an important characteristic of A* as it provides a bound on the amount of time that can be spent before the first plan is produced. Secondly, the algorithm has no control over the sub-optimality bound for all subsequent searches after the first – it only specifies the inflation factor, ϵ , for the first search and this inflation factor does not change in subsequent searches. To address these deficiencies, Likhachev, Gordon, and Thrun [15] developed the anytime repairing A* (ARA*) algorithm. This algorithm performs a series of weighed A* searches, where each subsequent search has a smaller inflation factor and reuses search efforts from previous searches. By specifying a smaller inflation factor in each subsequent search, the algorithm has gradually decreasing sub-optimality bounds for each subsequent search. The following discussion presents this algorithm.

2.5.1.4.3 Anytime Repairing A*

The anytime repairing A* (ARA*) algorithm works by running multiple weighed A* searches, starting with a large inflation factor, ϵ , and decreasing the inflation factor in each subsequent iteration until the final run in which $\epsilon = 1$. A large inflation factor corresponds to a search that will find a highly suboptimal solution quickly and an inflation factor of 1 corresponds to a search that is guaranteed to find the optimal path. For searches with inflation factors in-between, the algorithm is guaranteed to find a path that within a factor ϵ of optimal, and the sub-optimality tends towards the optimal solution with each subsequent search as ϵ is decreased and as it approaches 1. Again, a naive way to run multiple weighed A* searches is to run each weighed A* search from scratch. However, by reusing as much search effort from previous runs as possible, an efficient formulation can be realised. As opposed to running a weighted A* search from scratch in each run, ARA* saves computational efforts by reusing search efforts from previous runs.

Pseudocode for the algorithm is given in Algorithms 4 and 5. Algorithm 4 defines functions utilised by the main function whose pseudocode is shown in Algorithm 5. The `fValue` function (lines 1–2) is straight-forward, it simply computes the estimated cost to reach the goal via a given vertex. The task of the `improvePath` function is to recompute a path for a given inflation factor, ϵ . The main function then works by repetitively calling the `improvePath` function with a series of decreasing inflation factors.

Algorithm 4: Anytime Repairing A* – Part 1 (Auxiliary functions)

```

1 Function fValue(v)
2   return  $g(v) + \epsilon * h(v)$  ; // Use inflation factor to compute inflated f-value

3 Function improvePath()
4   while ( $fValue(v_G) > \min_{v \in OPEN} fValue(v)$ ) do
5      $v \leftarrow OPEN.getFirst()$  ; // Pop out vertex that is at the front of queue
6      $CLOSED \leftarrow CLOSED \cup \{v\}$  ; // Mark vertex as visited
7     foreach  $v' \in successors(v)$  do
8       if  $v' \notin CLOSED$  then
9          $g(v') \leftarrow \infty$  ; // Set g-values of unvisited successors to infinity
10      if  $g(v') > g(v) + c(v, v')$  then
11         $g(v') \leftarrow g(v) + c(v, v')$  ; // Update g-value of successor to new, lower value
12        if  $v' \notin CLOSED$  then
13           $OPEN \leftarrow OPEN \cup \{v'\}$  ; /* Mark successor for visitation if it has not been
14             visited */
15        else
16           $INCONS \leftarrow INCONS \cup \{v'\}$  ; /* Mark successor as inconsistent if it has been
17             visited */

```

The ARA* algorithm makes use of a notion known as *local inconsistency* that is described shortly. In A* search, when the g -value of a vertex, v' , is decreased as a result of it being a

Algorithm 5: Anytime Repairing A* – Part 2 (main function)

```

1 Function main()
2    $g(v_G) \leftarrow \infty$ ;  $g(v_I) \leftarrow 0$  ; // Initialisation of g-values of initial and goal vertices
3   OPEN  $\leftarrow \emptyset$ ; CLOSED  $\leftarrow \emptyset$ ; INCONS  $\leftarrow \emptyset$ ;  $\epsilon \leftarrow \epsilon_0$  ; // Initialise sets and inflation factor
4    $f(v_I) \leftarrow fValue(v_I)$  ; // Compute f-value of initial vertex
5   OPEN  $\leftarrow$  OPEN  $\cup \{v_I\}$  ; // Insert initial vertex into OPEN list
6   improvePath() ; // Compute new path based on the current inflation factor
7    $\epsilon \leftarrow \min(\epsilon, g(v_G)/\min_{v \in \text{OPEN} \cup \text{INCONS}}(g(v) + h(v)))$  ; // New inflation factor
8   publish current  $\epsilon$ -suboptimal solution;
9   while  $\epsilon > 1$  do
10    Decrease  $\epsilon$ ;
11    Move vertices from INCONS to OPEN;
12    Update priorities for all  $v \in$  OPEN according to  $fValue(v)$ ;
13    CLOSED  $\leftarrow \emptyset$  ; // Empty CLOSED list at the start of each new search
14    improvePath() ; // Compute new path based on the current inflation factor
15     $\epsilon \leftarrow \min(\epsilon, g(v_G)/\min_{v \in \text{OPEN} \cup \text{INCONS}}(g(v) + h(v)))$  ; // New inflation factor
16    publish current  $\epsilon$ -suboptimal solution;

```

successor to a vertex, v , which is being examined (line 15 of Algorithm 2), this decrease in the g -value of v' results in an inconsistency between the g -value of v' and the g -values of its successors. This inconsistency remains until v' is visited, at which point the g -values of its successors will be updated (lines 15 – 16 of Algorithm 2), correcting the inconsistency of v' . This correction, however, makes the g -values of the successors of v' inconsistent with respect to their successors. As a result, as the OPEN list is processed, local inconsistency is propagated to the successors of the processed vertices. As such, the OPEN list serves as the set of inconsistent vertices through which we propagate local inconsistency. Normal A* search (without an inflated heuristic) is guaranteed to visit each vertex at most once. With heuristic inflation, weighed A* loses this guarantee and it becomes possible for the algorithm to revisit vertices multiple times. ARA* implements a restriction that enforces the guarantee for a vertex to be only visited at most once in each weighed A* run. This restriction is implemented by checking vertices whose g -value has decreased and only inserting such a vertex into OPEN if and only if it was never visited (line 12–13 of Algorithm 4). All visited vertices are stored in the CLOSED list. While this strategy enforces the guarantee that each vertex is visited at most once in each weighed A* run, its consequence is that the OPEN list may no longer contain all the locally inconsistent vertices. Specifically, it will only contain the portion of locally inconsistent vertices that have not yet been visited. However, keeping track of all inconsistent vertices is necessary as they are needed for the propagation of inconsistency. To keep track of all inconsistent vertices (both visited and unvisited), the algorithm makes use of a set INCONS whose purpose is to store all inconsistent vertices that have been visited and hence not in OPEN (lines 14–15 of Algorithm 4). This means that the set of all locally inconsistent vertices is the set union of OPEN and INCONS. This set is then used as a starting point for the propagation of inconsistency prior to the start of each new search iteration. Besides the introduction of the INCONS set, another difference between improvePath and A* search is in their termination conditions. While a standard A* search terminates as soon as the goal vertex has been inserted into OPEN, since improvePath reuses search efforts from previous runs (i.e. not re-expanding vertices which were consistent in the previous search), then the goal vertex, v_G , may never become inconsistent in a certain run, and as such may never be inserted into OPEN. This makes the standard A* termination condition unsuitable for improvePath. A suitable termination condition is to stop the search as soon as the f -value of v_G becomes the smallest among all vertices in OPEN (line 4 of Algorithm 4). This termination condition also helps safeguard against expanding v_G or other vertices with an equal f -value.

As mentioned earlier, the main function of ARA* works by repeatedly calling the improvePath function with a series of decreasing inflation factors. Each call strives to produce a path with an improved cost compared to paths produced by previous calls. Prior to each improvePath call, the vertices that were in the INCONS list when the previous search terminated are moved into OPEN (line 11). Since the f -value of vertices in ARA* depends on the inflation factor, after

moving vertices from INCONS to OPEN, the f -values of the vertices in OPEN are recomputed using the new inflation factor (line 12). The sub-optimality bound is computed as the ratio of the g -value of the goal vertex and the minimum unweighed f -value of an inconsistent vertex. The new inflation factor is computed as the minimum between the current inflation factor and this ratio (line 10 and 18).

With the described mechanisms, the algorithm quickly finds an initial path like Anytime A*, but which, unlike anytime A*, is systematic and has iteratively decreasing inflation factors. Being systematic guarantees that the algorithm will not run indefinitely and the iteratively decreasing inflation factor ensures that each newly-found solution is cheaper. In the following subsection we move on to anytime replanning algorithms. Like the anytime planning algorithms of this subsection, they are able to find an initial path quickly, but unlike anytime planners they do so while also accounting for changes in the environment. Like incremental replanners, anytime replanners are outside the scope of this project and for this reason we only present the high-level ideas on which they are based, for the sake of completeness.

2.5.1.5 Anytime Replanning in Grid Costmaps

This subsection presents a high-level overview of grid-based anytime replanning algorithms. Like the incremental replanners of Subsection 2.5.1.3, anytime replanners are able to plan in environments that are allowed to change after path planning has begun. However, unlike incremental replanning (and like anytime path planning), anytime replanning handles situations where the planning problem is time-constrained.

To present an overview of how anytime replanners work, we consider an algorithm known as anytime dynamic A* (AD*) [16, 17] that was developed by Likhachev, Ferguson, Gordon, Stentz and Thrun, the developers of D* lite and ARA*. The AD* algorithm combines the replanning capability of D* lite with the anytime path planning capability of ARA*. Just like ARA*, it performs successive inflated A* searches, with each subsequent search having a smaller inflation factor than the previous one. The replanning capability inherited from D* lite comes into play when new information becomes known about the environment, which changes some edge costs. When this happens, vertices whose costs have changed are inserted into the OPEN list, so as to propagate the changes throughout the planning graph. The OPEN list is then processed until a path that is ϵ -suboptimal is found.

Just like ARA*, AD* begins with a large inflation factor, ϵ_0 . This is done to ensure that the initial, suboptimal solution is generated quickly. Once the initial, suboptimal solution is generated, the rest of the available planning time is spent improving the current solution or repairing portions of the current solution that have been invalidated by new information. Specifically, unless changes in the environment are detected, the inflation factor, ϵ , is reduced gradually, producing a series of improved solutions. This proceeds until the optimal path is found at $\epsilon = 1$. Each time ϵ is decreased and before the next search, all inconsistent vertices in INCONS are moved to OPEN and CLOSED is emptied. This behaviour is exactly the same as that of ARA*. However, whenever changes are detected in the environment, it is possible that the current path which was ϵ -suboptimal prior to these changes might not be ϵ -suboptimal any more. If this happens, replanning is necessary. Depending on the magnitude of the changes, the algorithm either repairs the current path to restore ϵ -optimality or increases ϵ to quickly find a less optimal replacement path. In particular, if the changes are significant, then repairing the current solution to regain ϵ -optimality might be a computationally expensive exercise or, in some cases, even infeasible, given the remaining planning time. In such cases, the algorithm relaxes the sub-optimality bound by increasing ϵ to quickly find a less optimal replacement path. While in ARA*, the only possible inconsistency is over-consistency; with AD*, under-consistency is also possible as edge costs change. As a result, vertices are inserted in the OPEN list with a key value which is the least between their old and new costs. Also, to ensure that the costs of under-consistent vertices are propagated to affected child vertices, their costs are calculated using the admissible, uninflated heuristic cost.

With the described mechanisms, the algorithm is able to handle both time-constrained path planning and changes in the robot's environment. This is the last technique for assumptive path planning in grid costmaps. The following subsection concludes our treatment of assumptive grid

costmaps by summarising our findings and evaluating the discussed algorithms against the path planning requirements of the project.

2.5.1.6 Conclusion on Assumptive Path-planning in Grid Costmaps

In this subsection, we reviewed algorithms that plan paths in grid costmaps without directly handling the uncertainty in cell costs, but rather making a simplifying assumption about the values of these costs. Different categories of planners were considered, namely, *optimal planners*, *incremental replanners*, *anytime planners* and *anytime replanners*. Each category of these planning algorithms is suited for specific situations. Particularly, optimal path planners are suited for situations in which the environment is fully known beforehand, and the planning problem is not time-constrained. Incremental replanners are suitable for situations in which the environment may change or is not fully known at the beginning of path planning, but the planning problem is not time-constrained. Anytime planners are useful in situations where the environment is static or fully known prior to the start of planning, but the planning problem is complex, yet time-constrained, and as such optimal planning may not be feasible in the available planning time. Lastly, anytime replanners are suitable for situations in which the path planning problem is time-constrained, and also, the environment is not fully known prior to start of planning or may change due to movement of obstacles.

The combinatorial path-planning algorithms discussed in Section 2.4 did not have this rich categorisation of path planners, capable of handling different situations. As such with combinatorial path planning, a chosen path-planning technique will behave exactly the same in all situations. This makes assumptive grid-costmap path planning more useful than combinatorial path planning. While these assumptive grid-costmap path planners have an edge over combinatorial planners and present interesting ideas for handling different situations, they have a number of limitations of their own with respect to the objectives of this project. Firstly, the discretisation of the configuration space into a grid brings back the curse of dimensionality associated with combinatorial path planners as the number of cells in the grid increases exponentially with an increase in dimension of the configuration space, making the approach to be very slow in high-dimensional configuration spaces – more so when a fine grid resolution is used. Secondly, these path planners rarely incorporate motion constraints – straight-line transitions between states are much more popular, resulting in paths that are concatenations of straight lines and hence discontinuous in curvature. While these paths can be post-processed through path smoothing techniques, the smoothed paths would no longer be guaranteed to be collision-free. Additionally, it is difficult, if at all possible, to prove that the smoothed paths are in fact executable by a vehicle with motion constraints. Even in the rare case that transitions that satisfy motion constraints are used, the resulting planner would still suffer from the curse of dimensionality. As such, these algorithms are not selected as suitable for the path planning aspect of this project. However, the interesting categorisation of path planners into the four discussed categories for different situations is considered and is shared with *sampling-based path-planning algorithms* discussed in Section 2.6 that were eventually selected for this project.

In the next subsection, we consider grid-based path planning under uncertainty which is also known as decision-theoretic path planning. While this approach was also not chosen in this project, it is important to discuss it for completeness of our review of existing path-planning techniques. Moreover, a decision-theoretic planner can coexist with the path planner implemented in this project, with the decision theoretic planner acting as a “user” in the autonomous navigation framework and providing the high-level user input, i.e. goal configurations, which is an input to the path planning and path optimisation submodule in the autonomous navigation framework.

2.5.2 Grid-based Path-planning Under Uncertainty

In the previous subsection, we reviewed grid-based path-planning techniques that avoided handling the uncertainty of cell costs directly by instead making a simplifying assumption that removes this uncertainty. The assumption essentially assigns a cost that is most probable for a particular cell as the actual cost for that cell, resulting in a scalar costmap as opposed to a probabilistic costmap. The assumptive approach has seen significant progress in the past two decades as evident from the range of planning algorithms discussed in the previous subsection and their applicability to

different types of situations. While this is the case, the assumption made by assumptive algorithms essentially discards potentially useful information by ignoring uncertainty. This subsection reviews grid-based path-planning techniques that takes uncertainty into consideration.

If uncertainty is considered in planning, then in choosing the next state to transition to, the robot must take into account all possible states that are consistent with its observations as the exact state is unknown [66]. Through the use of partially observable Markov decision processes (POMDPs) [67, 68], which handle uncertainty in a principled manner, it is possible to take into account, not only uncertainty in robot's states due to imperfect sensing, but also uncertainty in the effect of actions on states due to imperfect actuators and uncertainty about the environment. In a POMDP, the set of states is modelled as a probability distribution over the state space of the robot. This probability distribution is known as the belief, b . For a state, s , the probability, $b(s)$ gives the likelihood that the robot is in s . Solving a POMDP then involves reasoning in the belief space, \mathcal{B} – the space of all beliefs, computing a policy, π , which gives the best action for the robot to take for every belief that it may encounter. Thus, whereas other planners such as those discussed earlier and those that are yet to follow, compute a sequence of actions that the robot has to perform to reach the goal, POMDPs compute a universal policy for action selection. The robot then uses this policy to decide which action to take for any belief that may be encountered.

In POMDP terminology, the robot is referred to as a *decision maker* or an *agent*. This agent takes a sequence of decisions under uncertainty, applying a sequence of actions, over a number of time steps with the objective of maximising the total future reward. The problem in which such a decision maker has to learn how to act or behave when occasionally given rewards or punishments is also known as *reinforcement learning* (RL) [71, 72]. The problem is known as a *finite-horizon problem* if the number of time steps is finite, otherwise it is an *infinite-horizon problem*. Formally, a reinforcement-learning problem is specified by seven quantities, and denoted by (S, A, O, b_0, T, Z, R) . The first three quantities represent the following: S denotes a set of possible states, A denotes a set of possible actions and O is the set of possible observations. The next three quantities describe state transitions and perceived observations. They are introduced as part of the explanation of the operation of the agent. The last quantity describes the agent's performance and is also introduced as part of the explanation of the agent's operation that follows next.

The probability distribution, $b_0(s)$, is known as the *initial state probability distribution* and represents the probability of the agent being in state s at time $t = 0$. It is defined over all $s \in S$. At each time step, the agent picks an action $a \in A$, causing it to move from state $s \in S$ to another state $s' \in S$. Since there is uncertainty in robot control, the transition from s to s' after taking action, a , is modelled by a conditional probability function, $T(s, a, s') = p(s_t = s' \mid s_{t-1} = s, a_{t-1} = a)$, which gives the probability of the robot ending up in s' , after taking action a in state s . After executing action a from state s , the agent makes an observation and since there is uncertainty in observation, this observation is also modelled by a conditional probability function, $Z(s, a, o) = p(o_t = o \mid s_{t-1} = s, a_{t-1} = a)$, which gives the probability that the agent will make observation o after taking action a from state s . The reward function, $R(s, a)$, is used as an incentive for the agent to behave in a desirable manner and to penalise it for undesirable behaviour. It is a real-valued function and it is proportional to the utility of taking action a when in state s . The *discount factor*, $\gamma \in (0, 1)$, is useful for infinite-horizon problems since in such problems due to infinite number of time steps, the accumulated reward may tend towards infinity. Multiplying the reward per time step by the discount factor forces the accumulated reward to remain finite since it yields a geometric series that is guaranteed to converge to a finite value [25, p. 522]. In this infinite case, the accumulated reward, also known as the expected total reward, is given by $E[\sum_{t=t_0}^{\infty} \gamma^{t-t_0} R(s_t, a_t)]$, while in the finite case it is $E[\sum_{t=t_0}^T \gamma^{t-t_0} R(s_t, a_t)]$, where T represents the number of time steps over which we wish to evaluate the problem.

The task of a POMDP planner is to compute an optimal policy that maximises the expected total reward. POMDPs are a subclass Markov decision processes (MDPs). POMDPs differ from general MDPs in that in the general case, the agent's state is directly observable, while in the POMDP case it is only partially observable. This partial observability is result of observations $\{o_1, o_2, \dots, o_t\}$, which provide incomplete information about the state and consequently the state is not known exactly [66, 70]. Therefore, while in the general case, an MDP policy prescribes an action given the current state, i.e. $\pi : s_t \mapsto a_t$, a POMDP policy prescribes an action, given a

belief, i.e. $\pi : \mathcal{B} \mapsto \mathcal{A}$. Due to the partial observability of the state, a POMDP agent can select its actions based on its *experience* or *history*, h_t , which is a trace of all observations ever made and all actions ever executed. The history of an agent at the current time, t , is given by:

$$h_t = \{a_0, o_1, \dots, a_{t-2}, o_{t-1}, a_{t-1}, o_t\}. \quad (2.4)$$

The issue with keeping track of this history is that it can get very long as time goes on. A statistic that sufficiently summarises the history is the *belief distribution*, b_t , given by Equation 2.5. This represents the probability distribution over the states S as a vector with dimensions equal to those of S , where each element, $b_t(s_i)$, gives the agent's belief that it is in state s_i :

$$b_t(s) = p(s_t = s \mid o_t, a_{t-1}, o_{t-1}, \dots, a_0). \quad (2.5)$$

Apart from the fact that the belief distribution sufficiently summarises the history, it is also efficient to compute since the belief, b_t , at time t can be calculated and updated recursively using only the belief from one time step earlier, b_{t-1} , the most recent action a_{t-1} and observation, o_t . This computation is given by:

$$\begin{aligned} b_t(s) &= \tau(b_{t-1}, a_{t-1}, o_t) \\ &= \frac{\sum_{s'} Z(s', a_{t-1}, o_t) T(s, a_{t-1}, s') b_{t-1}(s')}{p(o_t \mid b_{t-1}, a_{t-1})}. \end{aligned} \quad (2.6)$$

It is based on Bayes rule. We refer to Equation 2.6 as the belief update equation and denote it by $\tau(b_{t-1}, a_{t-1}, o_t)$. With the ability to compute the agent's updated belief at any point in time according to belief update equation, a POMDP policy is of the form: $\pi(b) \mapsto a$, where the action, a , is the action chosen by the policy, π given a belief distribution b . When solving the POMDP problem, we are particularly interested in an *optimal policy* given in Equation 2.7, which maximises the expected future discounted cumulative reward:

$$\pi^*(b_t) = \operatorname{argmax}_{\pi} E_{\pi} \left[\sum_{t=0}^T \gamma^t r_t \mid b_t \right]. \quad (2.7)$$

To maximise the future reward, the policy induces a *value function*, V^{π} , which specifies the expected total reward for executing the policy, π , starting from the current belief, b_t . It is a mapping from belief states to real values. The optimal policy thus works by maximising this value function. The initial value function (i.e. at time $t = 0$), $V_0(b)$ is given by Equation 2.8:

$$V_0(b) = \max_a \sum_{s \in S} R(s, a) b(s). \quad (2.8)$$

The value function at time t can be computed from that of time $t - 1$ according to the recursive value function update equation given by Equation 2.9:

$$V_t(b) = \max_a \left[\sum_{s \in S} R(s, a) b(s) + \gamma \sum_{o \in O} p(o \mid a, b) V_{t-1}(\tau(b, a, o)) \right]. \quad (2.9)$$

While the value function of Equation 2.9 does maximise the expected sum of all future rewards, it only produces a real value and not an action, as a policy should. The value function thus fulfils the same objective as the optimal policy except that they have different return values. The policy can thus be extracted from the value function by simply returning the action that maximises the value instead of the value itself as:

$$\pi_t^*(b) = \operatorname{argmax}_a \left[\sum_{s \in S} R(s, a) b(s) + \gamma \sum_{o \in O} p(o \mid a, b) V_{t-1}(\tau(b, a, z)) \right]. \quad (2.10)$$

With a policy π given, the control of the agent's actions consists of two steps, executed repeatedly. The first step is known as policy execution. In this step, the agent takes an action $a = \pi(b)$, where b is its current belief. The second step is known as belief estimation, which is essentially a computation of a belief update according to Equation 2.6.

However, solving a POMDP (i.e. finding the optimal policy) is computationally intractable [73, 74] due to the curses of dimensionality and history. This is because finding such an optimal POMDP policy can be thought of as searching in a continuous belief space with dimension equal to the number of states in the robot's state space. To practically solve a POMDP, it is necessary to reduce the dimensionality of the belief space – in principle, searching a restricted belief space can be considerably easier [75–78]. Point-based POMDP algorithms [69, 70, 79, 80] achieve belief space reduction by sampling it probabilistically to produce approximate solutions. By so doing, point-based approaches extend the size of solvable POMDP problems such that they are useful for some robotics tasks. However, the state spaces considered by current point-based approaches tend to be in the order of 10×10 grids – a rather small state space [81]. As such, while promising, existing POMDP techniques remain computationally intractable for real-time planning in typical robotic navigation state spaces.

2.5.2.1 Conclusion on Grid-based Path-planning Under Uncertainty

The path-planning approach presented in this subsection handles uncertainty in a principled manner when planning in grid costmaps, unlike assumptive techniques which make a simplifying assumption to avoid planning under uncertainty. While planning under uncertainty in such a principled manner is without doubt beneficial, current methods remain intractable for real-time path planning. Apart from this shortcoming of the approach, due to being also grid-based, it also suffers from all shortcomings associated with assumptive planning in grid costmaps. For these reasons, this approach to path planning has not been selected as the suitable path-planning technique in this project.

2.5.3 Conclusion on Grid-based Path Planning

Grid-based path planning has been presented as an improved path-planning technique when compared to combinatorial path planning. The aspect that grid-based planners improve from combinatorial planners is that of getting rid of the unrealistic assumption of perfect knowledge of obstacle shape and position. Instead of making this assumption, grid-based planning overlays a grid on the configuration space and iterates over all grid cells with a classifier, assigning a cost to each cell. The resulting grid costmap is then used to search for a solution path. Two approaches for searching for solutions are namely assumptive planning and planning under uncertainty. Assumptive techniques make a simplifying assumption about the costs of uncertain cell costs so as to avoid handling uncertainty directly in the planning process. On the other hand, techniques for planning under uncertainty handle uncertainty in a principled manner. Assumptive techniques have greatly progressed over the past two decades and can be used for real-time path planning. Their drawback as far as this project is concerned is that they rarely produce paths that incorporate vehicle motion constraints. Even in the rare case that they do, they would still suffer from the curse of dimensionality, just like combinatorial path planners, since they discretise the configuration space. Techniques for planning under uncertainty on the other hand remain intractable for typical robotic navigation state spaces making them unsuitable for online path planning. They also suffer from the curse of dimensionality. The next section presents the last class of path-planning algorithms, namely, sampling-based path planners, which were eventually selected as suitable for the project.

2.6 Sampling-based Path Planning

Up to this point, we have considered two approaches to path planning in the configuration space, namely: combinatorial path planning and grid-based path planning. The combinatorial approach relies heavily on the exact knowledge of the position and shape of obstacles and it makes use of obstacle boundaries (vertices and edges) to construct a roadmap in $\mathcal{C}_{\text{free}}$ that can then be searched for a solution path. While planners based on this approach are exact and complete, they suffer from a number of issues, mainly the curse of dimensionality and the unrealistic assumption of perfect knowledge of obstacle shape and position. They are also unsuitable for this project as they do not allow for inclusion of vehicle motion constraints on the planned path. Grid-based path planners get rid of the unrealistic assumption of perfect knowledge of obstacles by instead

resorting to approximation of the obstacle region, \mathcal{C}_{obs} . However, they are still prone to the curse of dimensionality and also the produced paths do not incorporate vehicle motion constraints. In this section, we discuss sampling-based path-planning techniques that further improve on the planning capabilities of grid-based path planners by allowing the incorporation of vehicle motion constraints on the planned paths. They also do not suffer from the curse of dimensionality associated with both combinatorial and grid-based path planners. Like grid-based path planners, sampling-based path planners do not assume perfect knowledge of obstacle shape and position, but rather resort to approximating it. However, the way in which this approximation is achieved differs from that used by grid-based path planning, and it is explained in Subsection 2.6.1 which discusses the fundamentals of sampling-based path planning. Unlike both the combinatorial and grid-based path-planning approaches, sampling-based path planners provide an elegant way of incorporating vehicle motion constraints in planning. Due to these advantages over the other approaches, sampling-based path planning was selected as the suitable path-planning approach for the project. We now introduce the fundamentals of sampling-based path planning with reference to material from Chapter 5 and 13 of LaValle's book [25].

2.6.1 Fundamentals of Sampling-based Path Planning

Like grid-based path planners, and unlike combinatorial path planners, sampling-based path planners accept the fact that the environment cannot be perfectly known and therefore resort to approximating it. However the manner in which this approximation is done differs from that of grid-based path planning. Grid-based path planners discretise the configuration space, \mathcal{C} , by overlaying a grid on it and assigning a cost to each cell to approximate \mathcal{C}_{obs} . Planning then proceeds by first converting the grid into a graph and then performing a search on this graph to find a path from the initial configuration, \mathbf{q}_I , to the goal configuration, \mathbf{q}_G . On the other hand, the sampling-based path-planning approach applies a search strategy to probe \mathcal{C} directly (i.e. without first discretising it). The probing of the configuration space is done by using a sampling scheme and a collision detector to build a planning graph whose vertices or nodes correspond to reachable, collision-free configurations and whose edges represent collision-free paths connecting the reachable configurations. With the planning graph built, finding a path from the initial configuration to the goal configuration is achieved by searching this graph.

Figure 2.11 illustrates the principle behind a sampling-based path-planning algorithm. A sampling-based path planner integrates two ideas, namely, configuration space sampling and discrete search. The configuration space sampling aspect of the sampling-based path-planning approach involves choosing configurations to be added as nodes to the planning graph using a sampling scheme which might be either *deterministic* or *random*. The discrete search aspect of the approach pertains to exploration of the configuration space by growing a graph. The manner in which this graph is grown to explore the configuration space is strikingly similar to that of the discrete search techniques of Section 2.5.1 as explained shortly. In discrete search techniques, when adding a new vertex to the planning graph, an action that can be applied to reach the new vertex from an existing vertex on the graph is added as an edge between the new vertex and the existing (parent) vertex. Sampling-based path planning performs a similar operation when adding a new sample as a node on the planning graph. The difference is that instead of applying an action, sampling-based path planners generate a path that connects the new node to its parent. To be considered valid and added on the graph as an edge, this generated path has to be *feasible* and *collision-free*. A module of the sampling-based path-planning algorithm known as the *local path-planning method* (LPM), or the *local path planner*, is responsible for the generation of this path. It is this module that is tasked with ensuring that the generated path connecting a new node to its parent is feasible for the vehicle, i.e. that this path satisfies the vehicle's motion constraints. Following the name of the LPM, each path connecting a new node to its parent is called a *local path*. The usefulness of implementing the generation of local paths in a self-contained module of the sampling-based path planner is that if the vehicle model were to be changed, then only this module would have to be modified, with the rest of the sampling-based path planner remaining unchanged. The requirement for the generated local path to be collision-free is enforced by a separate module known as the *collision detector* whose purpose is to classify a candidate local path as collision-free or not. If collision-free, the local path can be added as an edge to the planning

graph built by the sampling-based path planner, otherwise it is discarded. The sampling-based path planner views the collision detector as a black box so as to separate path planning from the geometric model of the environment. Like in the case of the local path planner, the advantage of implementing collision detection as a self-contained module is that if the environment model (i.e. the map representation) were to change, then only the collision detection module would need to be modified, without altering the rest of the sampling-based path planner.

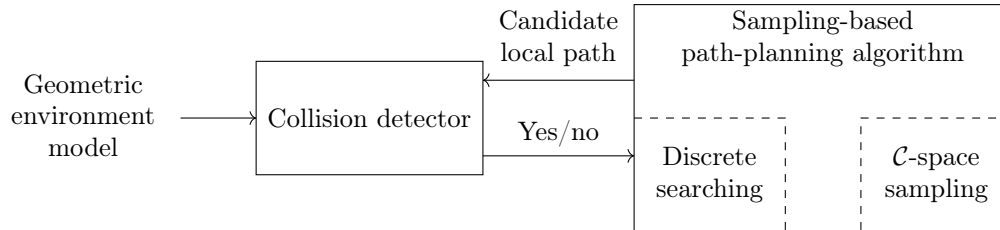


Figure 2.11: An illustration of the sampling-based path-planning approach (adapted from LaValle’s book [25]). A sampling-based path planner makes use of a sampling scheme and discrete search techniques to build a graph whose nodes are reachable configurations in \mathcal{C} and whose edges are feasible, collision-free paths between configurations. A collision detector is used to validate paths as collision-free before they are added as edges to the graph. This collision detector is viewed by the sampling-based path planner as a “black box”, making it possible to separate path planning from the particular geometric representation of the environment.

Like combinatorial and grid-based path planners, sampling-based path-planning algorithms can be classified according to completeness and exactness. As stated in the introduction, they are not exact, but rather approximate. In terms of completeness, like grid-based path planners, sampling-based path planners have a weaker notion of completeness that relies on a notion of density of samples. This notion is essentially based on the premise that samples come arbitrarily close to any configuration in $\mathcal{C}_{\text{free}}$ as the number of sampling iterations approaches infinity. There are two definitions of completeness for sampling-based path planners, depending on the nature employed sampling strategy. If a deterministic sampling strategy is used, then the resulting sampling-based path planner is said to be *resolution complete*. This term is the same as that which characterises the completeness of grid-based path planners; the difference is that in this case the resolution being referred to is sampling resolution and not grid resolution. Resolution completeness in this case means that the planner is guaranteed to find a solution, if one exists, if the sampling resolution is made fine enough. With a randomised sampling strategy, a sampling-based path planner is said to be *probabilistically complete*. This means that with the number of randomly sampled configurations approaching infinity, the probability that a solution path is found, if one exists, converges to one.

Sampling-based path-planning algorithms can be broadly divided into two categories, namely: *single-query* and *multi-query* algorithms. In the single-query case, a single configuration pair, $(\mathbf{q}_I, \mathbf{q}_G)$ is given as the path-planning query. In the multi-query case, many initial-goal configuration pairs are given as planning queries. Such a situation may arise when we have multiple robots operating in the same environment or if a single robot is required to navigate the same environment multiple times, solving a different planning query each time. Multi-query algorithms typically invest substantial pre-computational effort building a planning graph that can be later used to solve many planning queries. In contrast, for single-query settings, pre-computation offers no benefit since the planning query is only given once to a single robot. The problem being addressed by this thesis belongs to the single-query case. For brevity of our discussion, we do not discuss multi-query planners.

Algorithm 6, gives a general template on which most single-query sampling-based path-planning algorithms are based. We will occasionally refer to this template as we discuss different types of single-query sampling-based path planners in the rest of this section. Every sampling-based path planner begins by initialising the planning graph that it uses to probe the configuration space (line 1 of Algorithm 6). Initialisation constitutes adding the initial configuration, \mathbf{q}_I , as the only vertex

Algorithm 6: A general template for single-query sampling-based path planners.

Input: A single planning query $(\mathbf{q}_I, \mathbf{q}_G)$.
Output: A planning graph, \mathcal{G} , that strives to find a path from \mathbf{q}_I to \mathbf{q}_G .

```

1 initialisePlanningGraph() ; /* Add the initial configuration,  $\mathbf{q}_I$ , as a vertex on the search
   graph,  $\mathcal{G} = (V, E)$ . */
2 while termination condition not satisfied do
3    $\mathbf{q}_{\text{sampled}} \leftarrow \text{sampleConfig}()$  ; /* Sample a configuration in the configuration space. */
4    $\mathbf{q}_{\text{cur}} \leftarrow \text{selectVertex}(\mathcal{G}, \mathbf{q}_{\text{sampled}})$  ; /* Select an existing vertex,  $\mathbf{q}_{\text{cur}} \in V$ , on the graph
   from which to attempt extension towards  $\mathbf{q}_{\text{sampled}}$ . */
5    $\mathbf{q}_{\text{new}} \leftarrow$  a configuration that is determined based on how far we are willing to extend the
   graph towards  $\mathbf{q}_{\text{sampled}}$ ;
6    $\text{localPath} \leftarrow \text{planLocalPath}()$  ; /* Compute a candidate feasible path that connects  $\mathbf{q}_{\text{cur}}$ 
   to  $\mathbf{q}_{\text{new}}$ . */
7   if collisionFree(localPath) then
8     insertVertexAndEdge( $\mathbf{q}_{\text{cur}}$ ,  $\mathbf{q}_{\text{new}}$ , localPath) ; /* Add the local path to the edge set
    $E$ , marking it as an edge from  $\mathbf{q}_{\text{cur}}$  to  $\mathbf{q}_{\text{new}}$ . Also insert  $\mathbf{q}_{\text{new}}$  into the vertex set,  $V$ , if it
   is not already part of it. */
9     checkForSolution() ; /* Determine whether the planning graph,  $\mathcal{G}$ , contains a new
   solution path (i.e. a path reaching  $\mathbf{q}_G$ ) after the successful graph extension. */
10 return  $\mathcal{G}$ ;
```

in the vertex set, V , of the planning graph and initialising the graph's edge set, E , to an empty set to signify that no reachable configuration except \mathbf{q}_I has been found yet. With the planning graph initialised, the rest of the time is spent sampling configurations and attempting to extend the planning graph towards them. Particularly, in each iteration, a configuration, $\mathbf{q}_{\text{sampled}}$, is sampled (line 3) and an attempt to extend the graph towards this configuration is made. The attempt to extend the tree begins with the selection of a vertex, \mathbf{q}_{cur} , among existing graph vertices, from which the extension is to be attempted (line 4). Based on \mathbf{q}_{cur} and $\mathbf{q}_{\text{sampled}}$, a new configuration, \mathbf{q}_{new} , can be determined based on how far we are willing to extend the graph towards $\mathbf{q}_{\text{sampled}}$ (line 5). With this new configuration known, a local path can then be computed from the existing node, \mathbf{q}_{cur} , to the new node, \mathbf{q}_{new} (line 6). The computed local path needs to be checked for the absence of collisions before it can be added as an edge to the graph. If the local path passes the collision test (line 7), \mathbf{q}_{new} is added as a new node to the planning graph (if it is not already part of the graph's nodes) and the collision-free local path is added as an edge connecting \mathbf{q}_{cur} to \mathbf{q}_{new} (line 8); otherwise if the collision test fails, a new graph extension attempt is started. With a new edge added, a check can be performed to ascertain if the graph contains a new solution after the successful extension (line 9). If a new path has been found, it can be added to a set of found solutions. The process is repeated until a user-defined termination condition, such as the number of paths found, the maximum number of extension attempts or expiry of allocated planning time, is satisfied (line 2). On termination the planner returns the planning graph (line 10). From the planning graph, solution paths can be found by iterating through parent nodes, starting the goal configuration and moving backwards until the initial configuration is reached. If the goal configuration is not part of the returned planning graph, then no solution has been found.

Armed with the general framework for single-query sampling-based path planning, we now move on to discussing different types of single-query sampling-based path planners suited for different planning situations. We consider four categories, namely: *feasible* path planners which are the subject of the next subsection, *anytime* path planners which are discussed in Subsection 2.6.3, *optimal* path planners which we treat in Subsection 2.6.4 and *replanners* which are the subject of Subsection 2.6.5.

2.6.2 Single-query Sampling-based Feasible Path Planning

The problem of feasible path planning, as introduced in Subsection 2.6.1, is simply concerned with computing a collision-free path from the initial configuration to the goal configuration that satisfies the vehicle's motion constraints. No additional conditions such as constraints in planning time, the need to account for changes in the environment or requirements on the quality of the

path returned are placed on the feasible path-planning problem. It is thus the most basic form of path planning. In this subsection, we discuss single-query sampling-based path planners that are capable of solving this problem.

Since inception of the idea of sampling-based path planning, a number of single-query sampling-based path-planning algorithms capable of solving the feasible path planning problem have been introduced. These include randomised potential fields [83], Ariadne's clew algorithm [84], expansive-space planner [85], random walker [86, 87] and the rapidly-exploring random tree (RRT) [35, 89–92]. Among these, we focus our discussion on the most successful single-query sampling-based feasible path planner which is both quick and efficient in finding feasible, collision-free paths in complex obstacle-cluttered environments [36, 93–95], the *rapidly-exploring random tree* (RRT). Variants of this algorithm were eventually selected as suitable for the path planning requirement of this project.

The RRT path-planning algorithm was introduced by LaValle in 1998 [89]. As its name suggests, the planning graph that an RRT path planner builds to search for solution paths in the configuration space is a *tree*. This tree is rooted at the initial configuration and it is grown to *rapidly explore* the configuration space in its quest to find a solution path. The growth of the tree is guided by *densely* and *randomly* sampling the configuration space for the purpose of finding new configurations to be added as vertices. This results in a dense covering of the configuration space. The idea is that if a solution to the path planning problem exists, as the tree rapidly spreads through the configuration space, it will eventually reach the goal configuration, \mathbf{q}_G , or get arbitrarily close to it, at which point a solution is found. The RRT belongs to a larger class of dense tree-based path planners, known as *rapidly-exploring dense trees* (RDTs), which densely sample (either randomly or deterministically) the configuration space to grow a planning tree. The RRT is a subclass of RDTs for which the dense samples are drawn randomly. The use of randomisation helps the RRT to cope with uncertainty and to break the curse of dimensionality. It quickly searches high-dimensional spaces with algebraic constraints, imposed by the presence of obstacles, and differential constraints, imposed by the vehicle's motion constraints [91].

Algorithm 7: buildRRT($\mathbf{q}_I, \mathbf{q}_G$)

```

1  $K \leftarrow$  total number of iterations;
2  $\mathcal{T}.\text{initialize}(\mathbf{q}_I)$ ;
3  $p_{\text{goal}} \leftarrow$  goal bias ; // Probability of sampling goal configuration.
4 for ( $k = 1 : K$ ) do
5    $p \leftarrow \text{randomReal}([0.0, 1.0])$ ;
6   if ( $p \leq p_{\text{goal}}$ ) then
7      $\mathbf{q}_{\text{sampled}} \leftarrow \mathbf{q}_G$  ; /* Periodically sample goal so that a connection to it can be
      attempted. */
8   else
9      $\mathbf{q}_{\text{sampled}} \leftarrow \text{randomConfig}()$  ; // Sample a random configuration.
10   $\mathcal{T} \leftarrow \text{extendRRT}(\mathcal{T}, \mathbf{q}_{\text{sampled}})$  ; // Attempt tree extension towards the sample.
11 return  $\mathcal{T}$ ;
```

Algorithm 8: extendRRT($\mathcal{T}, \mathbf{q}_{\text{sampled}}$)

```

1  $\mathbf{q}_{\text{near}} \leftarrow \text{nearestNeighbour}(\mathcal{T}, \mathbf{q}_{\text{sampled}})$ ;
2  $\mathbf{q}_{\text{new}} \leftarrow$  a configuration that is one step size from  $\mathbf{q}_{\text{near}}$  and towards  $\mathbf{q}_{\text{sampled}}$ ;
3  $\text{localPath} \leftarrow \text{LPM}(\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}})$ ;
4 if ( $\text{collisionFree}(\text{localPath})$ ) then
5    $\mathcal{T}.\text{addVertex}(\mathbf{q}_{\text{new}})$ ;
6    $\mathcal{T}.\text{addEdge}(\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}}, \text{localPath})$ ;
```

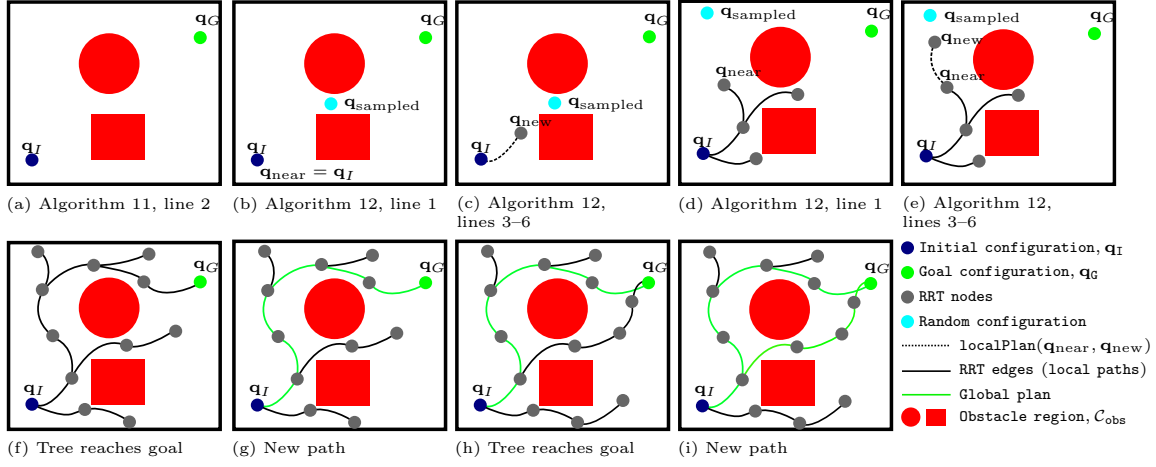


Figure 2.12: An illustration of the basic RRT path-planning algorithm.

The process of growing the tree is illustrated in Figure 2.12 with the algorithm's pseudocode given by Algorithms 7 and 8. We explain the working of the algorithm with reference to this pseudocode, the pseudocode of the general template for single-query sampling-based path-planning algorithms (Algorithm 6) and Figure 2.12. In line with the general template for single-query sampling-based path planning, the algorithm begins by inserting the initial configuration, q_I , as the only node on the tree (line 1 of Algorithm 6, line 2 of Algorithm 7 and Figure 2.12 (a)). After this initialisation, the rest of the planning time is spent growing the tree until it reaches the goal configuration, otherwise reporting failure if the allocated iterations run out without the goal configuration being reached. The tree is grown by iteratively sampling unexplored configurations from a uniform probability distribution of all possible configurations in \mathcal{C} . At each iteration, after such a sample ($q_{sampled}$) is generated (line 3 of Algorithm 6, Figure 2.12 (b) and (d), and lines 5–9 of Algorithm 7), an existing vertex, q_{near} , is selected that is nearest to $q_{sampled}$ (line 4 of Algorithm 6, Figure 2.12 (b) and (d), and line 1 of Algorithm 8). Once the existing node is selected for extension, a decision of how much to grow the tree from q_{near} and towards q_{rand} is then made based on a parameter known as the *step size*. This results in the computation of a configuration q_{new} that is a distance equal to the step size between q_{near} and q_{rand} (line 5 of Algorithm 6, Figure 2.12 (c) and (e), and line 2 of Algorithm 8). Once q_{new} has been determined, then an attempt to add it as a new node on the tree is made by determining the existence of a collision-free feasible local path from q_{near} to q_{new} . First, the local path is computed (line 6 of Algorithm 6, and line 3 of Algorithm 8). Then it is checked for collisions (line 7 of Algorithm 6, and line 4 of Algorithm 8). If the local path is found to be collision-free, it is added as an edge on the tree (line 8 of Algorithm 6, Figure 2.12 (c) and (e) and lines 5–6 of Algorithm 8). At this point, it is important to note the role of the goal-bias parameter, p_{goal} . This parameter (lines 3 and 6 of Algorithm 8) is used to focus (or bias) the RRT's search towards the goal – it causes the algorithm to sample the goal configuration (instead of a purely random configuration) occasionally, with probability p_{goal} , when extending the tree; with probability $1 - p_{goal}$, a purely random configuration is sampled. This results in an occasional attempt to connect the tree to the goal (i.e. an occasional attempt to find a solution path). Without goal bias, the RRT aggressively-explores unexplored regions of the configuration space, without any consideration of q_G [88]. By occasionally forcing $q_{sampled} \leftarrow q_G$ instead of sampling a random configuration, we introduce a gentle bias towards the goal in the algorithm and if in the process q_G is reached at any instance of this bias, then a solution path to the goal has been found. With the role of the goal bias explained, we now proceed with the last bit of our discussion of the algorithm. After each successful extension of the tree, if the goal configuration has been added to the tree during that extension (Figure 2.12 (f) and (h)), a new path is found (Figure 2.12 (g) and (i), and line 9 of Algorithm 6). The explained tree extension process is repeated until the number of allocated extension iterations expires. In the event that the allocated number of iterations runs out without the tree reaching the goal, then the algorithm

reports failure to find a solution. Alternatively, failure to find a solution path can be deduced by realising that the goal configuration is not part of the tree’s vertex set.

The first version of the RRT path-planning algorithm [89], made use of a straight-line LPM, i.e. when adding each node on the graph, the edge connecting it to an existing node on the tree is simply a straight line. As such, that version did not incorporate vehicle motion constraints. LaValle published a follow-up version that incorporates vehicle kinematics and dynamics a year later in 1999 [90], through the use of a kinodynamic LPM. Since then the RRT has been successfully used in numerous projects with different kinds of local path planners, giving concrete evidence that it is well-capable of planning paths that satisfy motion constraints.

At this point, we have introduced the rapidly-exploring random tree (RRT) as a single-query sampling-based feasible path planner that is capable of quickly and efficiently planning paths that satisfy motion constraints. It is important to reiterate that the RRT algorithm addresses the feasible path planning problem and does not attempt to achieve optimal path planning, i.e. it does not provide any bound on the costs of the paths it finds. In fact, in 2008, Nathan and Michael [95] described the costs of paths produced by RRT-based algorithms as “not well understood”. More concretely, a proof presented by Frazzoli and Karaman in 2010 [36] shows that the probability that the RRT algorithm converges to an optimal solution is zero. In the same paper, they introduced two variants of the RRT known as the *optimal RRT* (RRT^*) and the *k-nearest RRT**. They proved that these variants are asymptotically optimal, and that they retain the RRT’s characteristic of probabilistic completeness. Asymptotic optimality means that as the number of iterations tends to infinity, the probability that these variants find the optimal path tends to one. As such, these variants are alternatively said to “converge almost-surely to the optimum”. We defer further deliberation on the RRT^* and its variants to Subsection 2.6.4, where we discuss single-query sampling-based optimal path planning.

Though proven to be almost-surely suboptimal, path planning using the RRT can still be preferred in certain situations due to its simplicity, speed and efficiency. In such cases, one way to use the RRT for path planning and still get better paths is to make use of a path optimiser, based on the path-optimisation techniques presented in Section 2.8, to improve the path generated by the RRT before it is executed [88]. The second way by which the RRT can be used to produce better paths, though with no provable optimality guarantees, is the use of biasing heuristics to bias sampling or neighbour selection when extending the tree, so as to bias tree growth towards low-cost regions in the configuration space [36]. One notable piece of work that presents heuristics for biasing tree growth so as to improve the cost of found paths is that of Urmson and Simmons [97], which was published in 2003. In this work, they presented a number of heuristic functions including one which makes the likelihood of selecting an existing node on the tree for extension to depend on both the size of its Voronoi region and the cost of the path to that node. By increasing the weight of the cost of the path to a node when considering it for selection as a neighbour, the notion of a *good nearest neighbour* is introduced as opposed to just the nearest neighbour in the basic RRT. In essence, by favouring good nearest neighbours when growing the tree, exploitation of known good parts of the configuration space can be achieved. Conversely, by favouring nodes with large Voronoi regions, the algorithm can improve exploration in under-explored parts of the configuration space. By combining these ideas into a heuristic that considers both the cost of a node and its Voronoi region in vertex selection, the resulting hybrid heuristic can be used in balancing exploration and exploitation of the configuration space during tree growth. A third approach that can be used to produce better paths with the RRT, though without optimality guarantees, is a search-focusing technique known as the *anytime RRT*. It originates from grid-based anytime planning of Subsection 2.5.1.4 and was adapted to sampling-based path planning using RRT by Ferguson and Stentz [18]. The first way of improving an RRT path (i.e. doing so using a path optimiser) is in line with the investigation that this thesis aims to perform and it will be useful when we test our path optimisers after they have been developed. The second approach (i.e. using biasing heuristics), unfortunately, achieves solution quality improvement at a significant computational cost [18]. However, an idea originating from it, i.e. the notion of a good nearest neighbour, has been incorporated in anytime path planners (such as the anytime RRT) and optimal path planners (such as the RRT^*). Lastly, the third approach (i.e. improving paths produced by the RRT by employing an anytime path-planning scheme) has been successfully applied, even on improvements to optimal path planners like the RRT^* [20]. Due to this usefulness

of employing an anytime path-planning scheme, we discuss anytime path planning using the RRT in the next section.

2.6.3 Single-query Sampling-based Anytime Path Planning

As mentioned in Subsection 2.5.1.4, anytime path planning is suitable for situations where the planning problem is time-constrained. In these situations, it is important to quickly find an initial, possibly highly suboptimal, path and then spend the rest of the available time improving it. This subsection is focused on anytime path planning using RRTs.

In the context of RRT-based path planners, anytime path planning was introduced by Ferguson and Stentz [18], with inspiration from discrete anytime path-planning algorithms, particularly the anytime repairing A* (ARA*) algorithm of Subsection 2.5.1.4.3. To summarise, ARA* starts by computing an initial, possibly highly suboptimal, solution path using an inflated A* with a high inflation factor (or sub-optimality bound), ϵ . It then spends the rest of the available deliberation time improving the initial solution by running a series of new searches, with each successive search having a smaller sub-optimality bound than all previous searches. The cost of the path found by each search is guaranteed to be in the worst case ϵ times the cost of the optimal solution, otherwise less. This is the same basic idea that was used by Ferguson and Stentz to create the anytime RRT whose operation is explained next.

The anytime RRT starts by computing an initial, possibly highly suboptimal, path using an RRT without any cost consideration in vertex selection (i.e. the basic RRT). This is done so as to find the initial collision-free, feasible path quickly since the basic RRT is very quick and efficient in finding a feasible path. The cost of the path returned by this first RRT search, c_{best} – the cost of the best path found thus far – is then used to limit configurations added as nodes to a new subsequent RRT search to only those configurations that can possibly contribute to a new solution that is better than the best solution found so far. It is also possible to scale the cost bound, c_{best} , by a factor $(1 - \epsilon_i)$, where $0 \leq \epsilon_i < 1$ and ϵ_i is known as the *solution improvement factor*. This scaling of c_{best} by $(1 - \epsilon_i)$ ensures that the new solution, will be at least ϵ_i times cheaper than the previous one. This process is repeated until the allocated planning time runs out, with the cost of each newly-found solution appropriately scaled using the cost improvement factor and then used to focus the sampling of new configurations in the subsequent search. Again, the reduction in the sampled space encourages each subsequent search to find a better solution. It does not guarantee that a new solution will be found, but that if found, it will be better than all previous ones.

Apart from focusing the sampling of configurations considered for tree extension, with respect to the general template for single-query sampling-based path planning (Algorithm 6), the anytime RRT modifies line 4 (vertex selection method), and line 5 (the generation of the new configuration \mathbf{q}_{new} from the sampled configuration and the node selected for extension). We now discuss these introduced changes, starting with the sampling strategy.

Algorithm 9: Anytime RRT – part 1 (a) (Sampling strategy function definition)

```

1 Function sampleConfig( $c_{\text{best}}$ )
2    $p \leftarrow \text{randomReal}([0.0, 1.0]);$ 
3   if ( $p \leq p_{\text{goal}}$ ) then
4     return  $\mathbf{q}_G$ ;
5   else
6      $\mathbf{q}_{\text{sampled}} \leftarrow \text{randomConfig}();$ 
7      $\text{attempts} \leftarrow 0;$ 
8     while  $h(\mathbf{q}_I, \mathbf{q}_{\text{sampled}}) + h(\mathbf{q}_{\text{sampled}}, \mathbf{q}_G) > c_{\text{best}}$  do
9        $\mathbf{q}_{\text{sampled}} \leftarrow \text{randomConfig}();$ 
10       $\text{attempts} \leftarrow \text{attempts} + 1;$ 
11      if  $\text{attempts} > \text{maxAttemptsPerRRT}$  then
12        return null;
13    return  $\mathbf{q}_{\text{sampled}};$ 

```

Pseudocode for the sampling strategy of the anytime RRT is given in Algorithm 9. Like the sampling strategy of the basic RRT, it makes use of a goal-bias parameter to occasionally force the sampling of the goal configuration, with probability p_{goal} (lines 3–4), otherwise sampling a random configuration. The difference is that in this anytime case, the sampled random configuration (line 6) is examined to determine if it can possibly contribute to a cheaper solution than the current best solution before being accepted, otherwise it is discarded and a new sample is drawn (line 9). This process repeats until a suitable sample is found (line 8) or until the number of allowed sampling attempts is exhausted (line 11). The principle used to examine if a sampled configuration can possibly contribute to a better solution is based on the idea that for any configuration $\mathbf{q}_{\text{sampled}}$ in the configuration space, it is possible to calculate a heuristic cost for a path that starts from the initial configuration, \mathbf{q}_I , goes via $\mathbf{q}_{\text{sampled}}$ and ends at the goal configuration, \mathbf{q}_G . This heuristic cost (line 8) is similar to that used by the A* algorithm of Subsection 2.5.1.2.2. It is of the form:

$$f(\mathbf{q}_{\text{sampled}}) = h(\mathbf{q}_I, \mathbf{q}_{\text{sampled}}) + h(\mathbf{q}_{\text{sampled}}, \mathbf{q}_G), \quad (2.11)$$

where $h(\mathbf{q}_I, \mathbf{q}_{\text{sampled}})$ – also known as the heuristic cost-to-come – represents an estimate of the cost of the path from the root of the tree to $\mathbf{q}_{\text{sampled}}$, and $h(\mathbf{q}_{\text{sampled}}, \mathbf{q}_G)$ – also known as the heuristic cost-to-go – estimates the cost of the path from the sampled configuration to the goal configuration. As stated in Subsection 2.5.1.2.2, for such a heuristic to be considered admissible, it should not overestimate actual cost of the optimal path from \mathbf{q}_I to \mathbf{q}_G . This heuristic essentially gives a lower bound on the cost of any path that can start from \mathbf{q}_I , go via $\mathbf{q}_{\text{sampled}}$ and end at \mathbf{q}_G . As a result, if it happens for a given configuration, $\mathbf{q}_{\text{sampled}}$, that its heuristic cost exceeds the cost of the current best solution, c_{best} , then there is no way that $\mathbf{q}_{\text{sampled}}$ can be part of a solution that is cheaper than the current best solution, so $\mathbf{q}_{\text{sampled}}$ can be ignored.

Once a sample configuration, $\mathbf{q}_{\text{sampled}}$, has been drawn, a single-query sampling-based path planner has to select an existing node on the tree from which the tree extension is to be attempted (line 4 of Algorithm 6). The basic RRT simply selects an existing node that is closest to the sampled configuration. On the other hand, the anytime RRT uses ideas borrowed from Urmson and Simons [97] to incorporate cost considerations in this vertex selection step. Additionally, it uses *bias factors* to vary the influence of the cost of vertices and that of their proximity to the sampled configuration in being selected for extension. This bias is such that at one extreme (i.e. initially, with no path found) tree nodes are selected for extension purely based on their proximity to the sampled configuration (exactly like in the basic RRT). This makes the algorithm behave exactly like the basic RRT before the first path is found, enabling it to quickly find the initial path. Then in subsequent searches, a combination of proximity to $\mathbf{q}_{\text{sampled}}$ and the cost of nodes are considered for their selection. The bias factors are adjusted gradually with each new search, with the influence of cost increasing and that of proximity decreasing, such that at the other extreme eventually only the cost of nodes is considered for their selection. The explained vertex selection strategy is realised by first computing the k -nearest existing tree nodes to $\mathbf{q}_{\text{sampled}}$ and then ranking them according to a node selection function on each node \mathbf{q} . The node selection function is of the form:

$$\text{selCost}(\mathbf{q}) = c_b * c(\mathbf{q}_I, \mathbf{q}) + d_b * \text{distance}(\mathbf{q}, \mathbf{q}_{\text{sampled}}), \quad (2.12)$$

where c_b and d_b are known as bias factors, with c_b being the *cost bias factor* and d_b the *distance bias factor* – both bias factors are constrained to be in the range (0,1); $c(\mathbf{q}_I, \mathbf{q})$ is the cost of the path from \mathbf{q}_I to \mathbf{q} ; and $\text{distance}(\mathbf{q}, \mathbf{q}_{\text{sampled}})$ is the distance from \mathbf{q} to $\mathbf{q}_{\text{sampled}}$. Initially, with no path found, the bias factors are set to: $d_b = 1$ and $c_b = 0$. This is equivalent to nearest neighbour selection, as in the basic RRT. In each subsequent search after the initial path has been found, the distance bias, d_b , is reduced by some value δ_d and the cost bias c_b is increased by some value δ_c . This results in a gradual increase in the influence of the cost in vertex selection and a gradual decrease of the influence of proximity. The effect is that early on, costly solutions are found quickly, and then later on, cheaper solutions are produced if there is more planning time available.

With the existing tree node selected for extension, the next step in the general template for single-query sampling-based path planning is the generation of a new configuration, \mathbf{q}_{new} to be added as a new node on the tree (line 5 of Algorithm 6). Pseudocode for the extension procedure

of the anytime RRT is given in Algorithm 10. The basic RRT computes \mathbf{q}_{new} by moving a step-size from the selected node, \mathbf{q} , towards the sampled node, $\mathbf{q}_{\text{sampled}}$. Unlike the basic RRT, the anytime RRT does not consider only one extension from only one existing tree node; instead it considers multiple extensions from multiple (k nearest) existing tree nodes and then chooses the cheapest collision-free extension among them. This helps the algorithm to explore portions of the configuration space around each selected node, \mathbf{q} , and to generate low-cost tree extensions. In the pseudocode, the tree extension attempt begins with the computation of the k existing tree nodes that are closest to the sampled configuration (line 2). Then, each of these k neighbours is considered for extension, in ascending order of their node selection cost given by Equation 2.12 (line 4). For each node selected for extension, multiple extensions are considered and the cheapest collision-free one is accepted only if it satisfies the cost bound imposed by the current best path (line 8–9). Otherwise the remaining nodes are considered for extension. This process continues until a suitable extension is found, at which point, the new configuration to be added as a new node on the tree is returned (line 9) or until all k -nearest nodes have been considered in vain for extension. In the latter case, failure to find a suitable extension is reported (line 10).

Algorithm 10: Anytime RRT – part 1 (b) (Extension procedure function definition)

```

1 Function extendToSample( $\mathbf{q}_{\text{sampled}}$ )
2    $\mathbf{Q}_{\text{near}} \leftarrow \text{kNearestNeighbours}(\mathbf{q}_{\text{sampled}}, k, \mathcal{T})$ ;
3   while  $\mathbf{Q}_{\text{near}} \neq \emptyset$  do
4      $\mathbf{q}_{\text{neighbour}} \leftarrow \text{argmin}_{\mathbf{q} \in \mathbf{Q}_{\text{near}}} \text{selCost}(\mathbf{q}, \mathcal{T}, \mathbf{q}_{\text{sampled}})$  ; // Cheapest among neighbours.
5      $\mathbf{Q}_{\text{near}} \leftarrow \mathbf{Q}_{\text{near}} \setminus \mathbf{q}_{\text{neighbour}}$  ; // Pop-out cheapest neighbour.
6      $\mathbf{Q}_{\text{ext}} \leftarrow \text{generateExtensions}(\mathbf{q}_{\text{neighbour}}, \mathbf{q}_{\text{sampled}})$  ; /* Generate a number of possible
       collision-free extensions using LPM and collision detector. */
7      $\mathbf{q}_{\text{new}} \leftarrow \text{argmin}_{\mathbf{q} \in \mathbf{Q}_{\text{ext}}} c(\mathbf{q}_{\text{neighbour}}, \mathbf{q})$  ; // Choose cheapest extension
8     if  $c(\mathbf{q}_I, \mathbf{q}_{\text{new}}) + g(\mathbf{q}_{\text{new}}, \mathbf{q}_G) \leq c_{\text{best}}$  then
9       return  $\mathbf{q}_{\text{new}}$ ;
10  return null;

```

With a collision-free extension found, the next step in the general template for single-query sampling-based path planning (Algorithm 19) is the insertion of the new edge and node to the tree (line 8 of Algorithm 6). In the anytime RRT, this is fulfilled by the `growRRT` function outlined in Algorithm 11. This function keeps growing the planning tree until a solution with a cost satisfying the cost-bound imposed time is found, at which point the cost of the newly-found solution is returned (line 12) or until the allocated for each tree runs out. In the latter case, failure for the current tree search to find a solution is reported (lines 10–11). In each iteration of tree growth, a call to the `sampleConfig` function (Algorithm 9) is made (line 4), which returns a configuration towards which to attempt tree extension or failure to find such a sample (as explained earlier). If a valid sample is found, an attempt to generate an extension is made (lines 5–6) through a call to the `extendToSample` function (Algorithm 10), otherwise the `growRRT` proceeds to the next iteration of tree growth. The call to `extendToSample` returns the new configuration if the extension attempt was successful, otherwise reporting failure of the attempt. If a valid configuration is returned, the new configuration together with its local path is added to the tree (lines 7–8), otherwise `growRRT` proceeds to the next iteration of tree growth.

With the capability to grow an RRT tree that strives to produce a path whose cost is bounded from above in place, the main function of the anytime RRT simply coordinates the growth of successive search trees, each striving to get a better path than paths found by all previous ones. Pseudocode for the main function is outlined in Algorithm 12. It starts by initialising the parameters that guide the growth the successive trees; these are: the distance and cost bias factors (d_b and c_b) and the variable that holds the current best cost (c_{best}). Each of the successive trees is then built through the **repeat** loop (line 3). In each iteration of this loop, first, the planning tree from the previous (if any) is cleared. This clearing constitutes removing all nodes and vertices of the tree and then re-inserting the initial configuration. Both these operations are performed by the `reInitialiseRRT` function call (line 4). Then a tree striving to find a solution path that costs

Algorithm 11: Anytime RRT – part 1 (c) (growRRT function definition)

```

1 Function growRRT( $\mathcal{T}$ ,  $c_{\text{best}}$ )
2    $\mathbf{q}_{\text{new}} \leftarrow \mathbf{q}_I$ ; time  $\leftarrow 0$ ;
3   while distance( $\mathbf{q}_{\text{new}}$ ,  $\mathbf{q}_G$ ) > goalThreshold do
4      $\mathbf{q}_{\text{sampled}} \leftarrow \text{sampleConfig}(c_{\text{best}})$ ;
5     if ( $\mathbf{q}_{\text{sampled}} \neq \text{null}$ ) then
6        $\mathbf{q}_{\text{new}} \leftarrow \text{extendToSample}(\mathbf{q}_{\text{sampled}}, \mathcal{T})$ ;
7       if ( $\mathbf{q}_{\text{new}} \neq \text{null}$ ) then
8          $\mathcal{T}.\text{addNode}(\mathbf{q}_{\text{new}})$ ;
9       updateTime(time);
10      if (time > maxTimePerRRT) then
11        return null;
12  return  $c_{\text{best}}$ ;
    
```

less than c_{best} is grown through the growTree function (line 5). When a better solution is found, the cost of the current best path is accordingly updated and the solution is published so that it can be executed if the robot is required to act. The bias factors are also updated by gradually increasing the cost bias and reducing the distance bias as explained earlier. This process is repeated until planning time runs out. The result is an RRT-based path-planning algorithm capable of planning in anytime fashion, i.e. quickly finding solutions that are possibly highly suboptimal earlier on, when it is expedient to at least have a feasible path to the goal (no matter the cost), and then producing better replacement solutions as planning time permits later on.

Algorithm 12: Anytime RRT – part 2 (main function definition)

```

1   $c_b \leftarrow 0$ ;  $d_b \leftarrow 1$ ;  $c_{\text{best}} \leftarrow \infty$ ; // Initialisation of variables.
2  Function main()
3    repeat
4      reInitialiseRRT( $\mathcal{T}$ ) ; // Clear tree and update root.
5       $c_{\text{new}} \leftarrow \text{growRRT}(\mathcal{T}, c_{\text{best}})$ ;
6      if  $c_{\text{new}} \neq \text{null}$  then
7        postCurrentSolution( $\mathbf{Q}_{\text{soln}}$ ) ; // Publish the current best solution.
8         $c_{\text{best}} \leftarrow c_{\text{new}}$  ; // Update cost of current best solution.
9         $c_b \leftarrow c_b + \delta_c$  ; // Update cost bias (increase it gradually as tree grows).
10       if  $c_b > 1$  then
11          $c_b \leftarrow 1$  ; // Prevent cost bias from getting greater than 1.
12        $d_b \leftarrow d_b - \delta_d$  ; // Update distance bias (increase it gradually).
13       if  $d_b < 1$  then
14          $d_b \leftarrow 0$  ; // Prevent distance bias from being negative.
15  until planningTimeElapsed();
    
```

While the anytime RRT, as described, is able to quickly find an initial path and to improve it with additional available planning time, it does not guarantee that the series of improvements will eventually reach the optimal solution. The next subsection discusses RRT-based path planners that are guaranteed to find the optimal path.

2.6.4 Single-query Sampling-based Optimal Path Planning

As stated in the Subsection 2.6.1, the problems of feasible and optimal path planning differ in that feasible path planning is only concerned with computing a collision-free path satisfying vehicle motion constraints without regard of the cost of executing this path. Optimal path planning on the other hand is concerned with finding a collision-free feasible path with minimal cost. It was also stated that the basic RRT algorithm addresses feasible path planning and that the earliest

variant of the RRT to address optimal path planning was the RRT*, and that this algorithm attains optimality in an asymptotic manner and as such is said to be asymptotically optimal. Optimal RRT-based path-planning algorithms that come after the introduction of the RRT* aim to achieve optimality in finite time and to speed-up the rate of convergence to the optimal solution. Notable successors of the RRT* that have made progress in this regard include the RRT*-smart, the informed RRT* and the wrapping-based informed RRT*. In this subsection, we review algorithms for solving the problem of single-query optimal path planning using RRTs. Our review starts with the RRT* as the earliest optimal path planner based on the RRT and proceeds to the notable successors of the RRT* to date.

2.6.4.1 The Optimal RRT (RRT*)

The RRT* is based on the rapidly-exploring random graph (RRG) algorithm that was developed in 2010 by Karaman and Frazzoli [99] – the same authors as the RRT*. They initially proposed the RRG in 2009 [100]. Having proved that the solutions returned by RRT-based algorithms converge almost-surely to a non-optimal solution, Frazzoli and Karaman introduced the RRG as an algorithm that converges to the optimal solution almost-surely. The RRG algorithm works in a similar way as the RRT in that it first tries to connect the new node to the nearest node on the graph and if the connection does not result in collisions with obstacles, the new node is added to the vertex set and the local path connecting the nodes is added to the edge set. It is different from the RRT in that it goes an extra step when adding a new node on the graph. This extra step involves attempting connections from all other nodes in the vertex set that are within a ball radius to the newly added node, adding an edge for each successful connection. As such, for the same sampling sequence, the planning graphs constructed by the RRT and the RRG have exactly the same vertex set; the only difference is their edge sets, with the edge set of the RRT being a subset of that of the RRG. Due to the difference in the edge sets, the structure of the graphs constructed by the two algorithms also differs. Since in the RRT case, only one edge is constructed for each node, the graph structure is a directed tree. However, for the RRG, since there are possibly multiple edges reaching a given node, the graph structure is an undirected graph, similar to a roadmap and it possibly contains cycles. The roadmap constructed by the RRG is built incrementally online and for single-query purposes as opposed to that built by multi-query planners like the Probabilistic Roadmap (PRM) [159] that is batch-constructed offline and for multi-query purposes.

The RRT* algorithm then results from a modification of the RRG in a way that removes cycles in the graph by removing redundant edges for each node. Only the edge that results in the shortest path from the root to each node is added to the planning graph. This results in the planning graph being a directed tree just like in the RRT case. Also, like the RRT, for the same sampling sequence, the vertex set of the RRT* is exactly the same as that of the RRG and the edge set of the RRT* is a subset of that of the RRG. The first difference between the construction of the RRG and the RRT* is in that when adding a new node, the RRT* considers the neighbours that are within a ball radius to the new node and only connects the new node through a neighbour that results in a minimum-cost path from the root, discarding all other possible connections. This is in contrast to the RRG which makes all possible connections from neighbours lying within the ball radius to the new node. The second difference lies in an additional step known as *rewiring* that the RRT* performs after adding the new node. This step considers the newly added node as a replacement parent for each of its neighbours that are within the ball radius and replaces the edge from each existing parent node with an edge from the new node if the connection from the new node to the neighbour has a lower cost than the connection from the existing parent. The algorithm's pseudocode is shown in Algorithm 13. Lines 1–2 initialise the tree by inserting the initial configuration to the vertex set and initialising the edge set to an empty set. New nodes are generated (lines 4–8) and added (line 9) the same way as in the RRT to the vertex set. However, the computation of edges differs as explained earlier. In this case, connections are considered from existing vertices that are within a ball of radius r_{RRT^*} from the new configuration (line 8). Not all of the considered connections result in a new edge; instead the one resulting in a minimum cost path among the neighbours results in the new connection (lines 10–17). Once the new node and its edge are added to the tree, the rewiring process is performed. Through this process, new edges are created from \mathbf{q}_{new} to nodes in \mathbf{q}_{near} if the path to the concerned node through \mathbf{q}_{new} has

a lower cost than the path to the node through its existing parent (lines 18–23). The edge from the existing parent is then deleted (line 24) so as to maintain a tree structure.

Algorithm 13: RRT*($\mathbf{q}_I, \mathbf{q}_G$)

```

1  $V \leftarrow \{\mathbf{q}_I\}; E \leftarrow \emptyset;$ 
2  $\mathcal{T} \leftarrow (V, E);$ 
3 for iteration  $\leftarrow 1$  to maxIterations do
4    $\mathbf{q}_{\text{rand}} \leftarrow \text{randomSample}();$ 
5    $\mathbf{q}_{\text{nearest}} \leftarrow \text{nearestNeighbour}(\mathcal{T}, \mathbf{q}_{\text{rand}});$ 
6    $\mathbf{q}_{\text{new}} \leftarrow$  a configuration that is one step size from  $\mathbf{q}_{\text{nearest}}$  and towards  $\mathbf{q}_{\text{rand}};$ 
7   if collisionFree(LPM( $\mathbf{q}_{\text{nearest}}, \mathbf{q}_{\text{new}}$ )) then
8      $\mathbf{Q}_{\text{near}} \leftarrow \text{near}(\mathcal{T}, \mathbf{q}_{\text{new}}, r_{\text{RRT}^*});$ 
9      $V \leftarrow V \cup \{\mathbf{q}_{\text{new}}\};$ 
10     $\mathbf{q}_{\text{min}} \leftarrow \mathbf{q}_{\text{nearest}};$ 
11     $c_{\text{min}} \leftarrow \text{cost}(\mathbf{q}_{\text{min}}) + \text{localPathCost}(\mathbf{q}_{\text{min}}, \mathbf{q}_{\text{new}});$ 
12    foreach  $\mathbf{q}_{\text{near}} \in \mathbf{Q}_{\text{near}}$  do
13       $c_{\text{new}} \leftarrow \text{cost}(\mathbf{q}_{\text{near}}) + \text{localPathCost}(\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}});$ 
14      if collisionFree(LPM( $\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}}$ )) and  $c_{\text{new}} < c_{\text{min}}$  then
15         $\mathbf{q}_{\text{min}} \leftarrow \mathbf{q}_{\text{near}};$ 
16         $c_{\text{min}} \leftarrow c_{\text{new}};$ 
17     $E \leftarrow E \cup \{\text{LPM}(\mathbf{q}_{\text{min}}, \mathbf{q}_{\text{new}})\};$ 
18    foreach  $\mathbf{q}_{\text{near}} \in \mathbf{Q}_{\text{near}}$  do
19       $c_{\text{near}} \leftarrow \text{cost}(\mathbf{q}_{\text{near}});$ 
20       $c_{\text{new}} \leftarrow \text{cost}(\mathbf{q}_{\text{new}}) + \text{localPathCost}(\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{near}});$ 
21      if collisionFree(LPM( $\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{near}}$ )) and  $c_{\text{new}} < c_{\text{near}}$  then
22         $\mathbf{q}_{\text{parent}} \leftarrow \text{getParent}(\mathbf{q}_{\text{near}});$ 
23         $E \leftarrow E \cup \text{LPM}(\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{near}});$ 
24         $E \leftarrow E \setminus \text{LPM}(\mathbf{q}_{\text{parent}}, \mathbf{q}_{\text{near}});$ 

```

While the RRT* algorithm described above succeeds in finding an optimal path, it does so asymptotically and in the process of finding this optimal solution, the algorithm finds the optimal path from the initial configuration to every configuration in the planning domain [20]. This behaviour is similar to that of Dijkstra’s algorithm that was discussed in Subsection 2.5.1.2.1. As such, the algorithm is inefficient. Moreover, the algorithm’s characteristic of finding optimal paths from the initial configuration to every configuration in the planning domain is inconsistent with its single-query nature. We now outline algorithms presented as improvements in an attempt to address the mentioned issues with the RRT*.

2.6.4.2 Notable Successors of the RRT*

A way to reduce the time taken by the RRT* to converge to the optimal solution and to prevent it from optimising paths to all configurations in the planning domain is to limit the sampled configurations to only those that can possibly improve the current solution. This can be seen as focusing the search process. If we let $f(q)$ denote the cost of an optimal path from \mathbf{q}_I to \mathbf{q}_G that is constrained to pass through the configuration q and $\mathcal{C}_f \subseteq \mathcal{C}$ denote the subset of configurations that can possibly improve the current solution, that is:

$$\mathcal{C}_f = \{q \in \mathcal{C} \mid f(q) < c_{\text{best}}\}, \quad (2.13)$$

where c_{best} is the cost of the current best solution, then essentially, focusing the search is equivalent to increasing the probability that new configurations added to the tree belong to \mathcal{C}_f . Various focusing strategies have been considered in literature, namely *sample biasing*, *heuristic-based sample rejection*, *graph pruning* and *anytime planning* [20].

Sample biasing increases the probability that sampled configurations belong to \mathcal{C}_f by biasing the distribution of samples drawn from \mathcal{C} . Issues with sample biasing include the fact that such biasing does not eliminate sampling of configurations outside \mathcal{C}_f , but rather minimises its likelihood. Secondly, biasing the sampling results in a non-uniform density over the sampled space – a violation of a key assumption of the RRT*. Two strategies for sampling biasing exist, namely heuristic-based sampling and path biasing. Path biasing tries to increase the probability of sampling configurations belonging to \mathcal{C}_f by sampling configurations that lie in the vicinity of the current solution path. One improvement on the RRT* based on path biasing is the work by Akgun and Stilman [101], where the algorithm randomly selects a configuration on the current solution path and then explicitly samples random configuration from the selected configuration's Voronoi region. Another improvement on the RRT* that is based on path biasing is the RRT*-Smart algorithm developed by Nasir et. al. [94]. For each found solution path, their algorithm improves it by first smoothing this path so that it contains a minimum number of nodes. It then uses the remaining nodes on the path to bias the sampling of new configurations by sampling within ball radii to these path nodes. In addition to the aforementioned issues with sample biasing, a unique issue with path biasing is that since it focuses the search near the current solution, it may reduce the probability of finding a path of a different homotopy class.

Heuristic-based sample rejection tries to increase the probability of sampling configurations belonging to \mathcal{C}_f by using sampling rejection on \mathcal{C} . Samples are drawn from a larger distribution that is easier to characterise and which is known to contain the actual informed subset such as a rectangle or hyper-rectangle (as in Otte and Correl's parallelised C-Forest algorithm [102]). The drawn samples are either kept or rejected based on their heuristic cost. While sample rejection does result in focusing the search, it becomes inefficient as the solution approaches the theoretical optimal or when the size of the sampling space is large.

The third technique for focusing the search, i.e. graph pruning, works by periodically removing nodes on the planning graph that are outside \mathcal{C}_f – i.e. those whose heuristic cost from the root to the goal is greater than the cost of the current solution. Graph pruning was used by Karaman et. al. [19] in their implementation of an anytime version of the RRT*. They used the current cost of a node as the cost-to-come and estimated the cost from the node to the goal (i.e. cost-to-go) using a heuristic. However, using the node's current cost as the cost-to-come in computing its heuristic cost may possibly overestimate the node's cost and is as such an inadmissible heuristic [20]. As such, this may lead to erroneous removal of configurations that in fact belong to \mathcal{C}_f simply because their current cost-to-come is not yet optimal, since in the RRT*, the cost-to-come of each node approaches its optimal value from above.

The last strategy for focusing the RRT* search is anytime planning. It originates from grid-based anytime planning of Subsection 2.5.1.4 and was adapted to sampling-based path planning by Ferguson and Stentz [18] with application to RRTs, as discussed in Subsection 2.6.3. In their formulation, the algorithm is iterative; it solves a series of independent RRT searches, with the domain of each subsequent search bounded by the cost of the previous solution and represented as an ellipse. As such, the sampling space reduces with each better solution found and this reduction in the sampling space encourages each subsequent search to find a better solution. Each subsequent search starts searching from scratch and as such discards configurations belonging to \mathcal{C}_f which were found by previous searches. This principle of anytime path planning is employed by the informed RRT*, which focuses the RRT* search and accelerates its rate of convergence, without suffering from any of the issues pointed out with the other search-focusing strategies. The informed RRT* is discussed next.

2.6.4.3 The Informed RRT*

The informed RRT* is an improvement of the RRT* that aims to speed up its rate of convergence to prevent it from improving paths to configurations that cannot possibly improve the current solution while overcoming the weaknesses of the four focusing strategies discussed above. Unlike heuristic sample biasing, it does not sample configurations that cannot possibly improve the current solution and unlike path biasing, it does not make an assumption about the homotopy class of the optimal solution. Also, unlike both heuristic sample biasing and path biasing it does not violate the RRT*'s key assumption of a uniform density of samples. Moreover, unlike heuristic-based rejection

sampling, it remains effective even when the size of the sampling space increases and even when the cost of the current solution approaches the theoretical minimum. Lastly, unlike anytime RRTs and graph pruning, it does not at any point discard configurations belonging to \mathcal{C}_f that have been found, but is able to keep all such configurations for the duration of the planning process, hence reusing as much effort from previous searches as possible.

Just like the A* algorithm of Subsection 2.5.1.2.2, which was formulated to address similar issues with Dijkstra's algorithm, the informed RRT* focuses the search of the RRT* using an admissible heuristic. In this way, the algorithm only considers those configurations which can possibly improve the current solution. Through this modification, the informed RRT* improves the rate of convergence to the optimal solution while retaining the RRT*'s completeness and optimality guarantees.

Similarly to assumptive planning in grid costmaps, the heuristic estimate of the optimal cost of the path from the root, \mathbf{q}_I , to the goal, \mathbf{q}_G , constrained to pass through a node, \mathbf{q} , in the RRT* tree can be denoted by $f(\mathbf{q})$, and called the f -value. It is the sum of an estimate of the cost of the optimal path from the root to \mathbf{q} , which is known as the heuristic cost-to-come, denoted by $h(\mathbf{q})$, and an estimate of the cost of the path from \mathbf{q} to the goal known as the heuristic cost-to-go, denoted by $g(\mathbf{q})$. A necessary condition is for f -value to be admissible, i.e. that it should never overestimate the true optimal cost of the node. An implication of this requirement is that the h and g -values should be individually admissible. The informed RRT* uses the Euclidean distance from the root to \mathbf{q} as an admissible cost-to-come heuristic and the Euclidean distance from \mathbf{q} to the goal as an admissible cost-to-go. This is because the Euclidean distance is an admissible heuristic for problems seeking to minimise path length, even with motion constraints [20]. The *informed* subset of configurations which can possibly improve the cost of the current solution, c_{best} , is given by:

$$\mathcal{C}_f = \{\mathbf{q} \in \mathcal{C} \mid \|\mathbf{q}_I - \mathbf{q}\|_2 + \|\mathbf{q} - \mathbf{q}_G\|_2 \leq c_{\text{best}}\}. \quad (2.14)$$

The region bounding the configurations which satisfy Equation 2.14 is an ellipsoid whose focal points are \mathbf{q}_I and \mathbf{q}_G , with traverse diameter, c_{best} and conjugate diameters equal to $\sqrt{c_{\text{best}}^2 - c_{\text{min}}^2}$, where c_{min} is the theoretical minimum cost between the initial and goal configurations. It is illustrated in Figure 2.13. By directly sampling this subset of the configuration space, the informed RRT* improves the rate of convergence to the optimal solution while retaining the RRT*'s completeness and optimality guarantees. Figure 2.14 shows example runs of the RRT* and the informed RRT*. The algorithms were run until they each found a solution of the same cost. From a number of reported results [20], of which the result shown in this figure is part of, the informed RRT* clearly accelerates the rate of convergence of the RRT* significantly while retaining its ability to converge to the optimal path.

Pseudocode for the informed RRT* algorithm is given in Algorithm 14. Line 1 initialises the search tree as well as the set used to store found solution paths and the cost of the current best solution. Tree growth occurs within the *for* loop in lines 2–11. Besides the focusing of the sampling process using the cost of the current best solution, the growth of the informed RRT* search tree is exactly the same as that of the RRT*. The informed RRT* samples random configurations from the entire configuration space, just like the RRT* (line 7), until the first solution path is found.

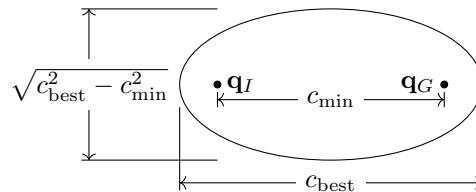


Figure 2.13: The region sampled by the informed RRT* is an ellipse whose focal points are \mathbf{q}_I and \mathbf{q}_G , and whose eccentricity is given by $c_{\text{min}}/c_{\text{best}}$. Here, c_{min} is the theoretical minimum cost path between \mathbf{q}_I and \mathbf{q}_G , and c_{best} is the cost of the current best solution. Adapted from Gammell et al. [20].

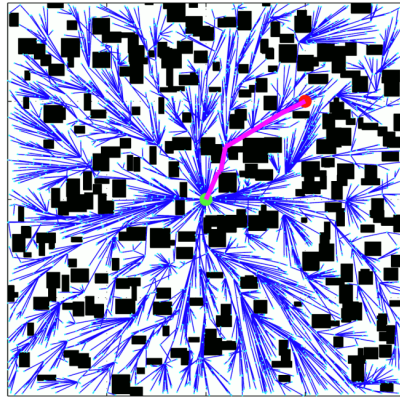
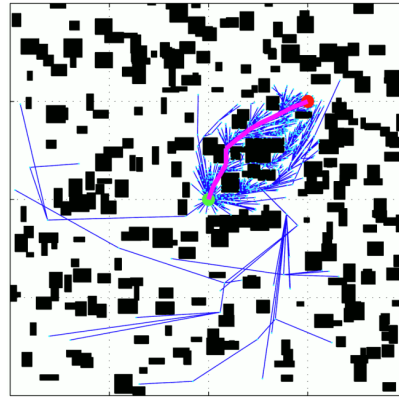
(a) RRT*, 8.26 seconds, $c_{\text{best}} = 0.76$ (b) Informed RRT*, 1 second, $c_{\text{best}} = 0.76$

Figure 2.14: Example runs of the RRT* and the informed RRT* for the same path planning problem. Through informed sampling, the informed RRT* significantly accelerates the rate of convergence towards the optimal solution and reduces unnecessary exploration of parts of the configuration space that cannot contribute to the optimal solution. Sourced from Gammell et al. [20].

Thereafter, the algorithm uses the cost of the current best solution to draw a sample from the subset of the configuration space containing configurations that can possibly improve the current solution (line 4). For this reason, to avoid repetition, we abstract the operations performed by the RRT* when extending and rewiring a tree, \mathcal{T} , after a sampled configuration, $\mathbf{q}_{\text{sampled}}$, has been drawn. This abstraction is realised through the function `extendAndRewireRRT*` in our informed RRT* pseudocode. With this abstraction in place, once a sample has been drawn by the informed RRT* (line 4 and 7), we simply call `extendAndRewireRRT*`, passing the sampled configuration as a parameter (lines 5 and 8). If tree extension and rewiring results in a new solution being found, such a solution is added to the set of found solutions (line 10). The current best path cost c_{best} is updated (line 11) and will be used to further focus the sampling of configurations in the subsequent search. This process repeats until the allocated number of iterations elapses.

Among the considered successors of the RRT*, the informed RRT* provides an improvement that significantly improves the efficiency of the RRT* while retaining its probabilistic completeness and optimality guarantees. While the informed RRT* provided such a significant improvement, Kim and Song [98] proposed an improvement to the informed RRT*, which is known as the *wrapping-based informed RRT** to further accelerate the rate of convergence. Their algorithm is discussed next.

Algorithm 14: `informedRRT*($\mathbf{q}_I, \mathbf{q}_G$)`

```

1  $V \leftarrow \{\mathbf{q}_I\}; E \leftarrow \emptyset; \mathcal{T} \leftarrow (V, E); \mathbf{Q}_{\text{sol}} \leftarrow \emptyset; c_{\text{best}} \leftarrow \infty;$ 
2 for iteration  $\leftarrow 1$  to maxIterations do
3   if  $\mathbf{Q}_{\text{sol}} \neq \emptyset$  then
4      $\mathbf{q}_{\text{sampled}} \leftarrow \text{informedSample}(\mathbf{q}_I, \mathbf{q}_G, c_{\text{best}});$ 
5      $\mathcal{T} \leftarrow \text{extendAndRewireRRT}^*(\mathcal{T}, \mathbf{q}_{\text{sampled}});$ 
6   else
7      $\mathbf{q}_{\text{sampled}} \leftarrow \text{randomSample}();$ 
8      $\mathcal{T} \leftarrow \text{extendAndRewireRRT}^*(\mathcal{T}, \mathbf{q}_{\text{sampled}});$ 
9   if new solution found then
10     $\mathbf{Q}_{\text{sol}} \leftarrow \mathbf{Q}_{\text{sol}} \cup \text{new solution};$ 
11     $c_{\text{best}} \leftarrow \text{cost of the new solution};$ 

```

2.6.4.4 The Wrapping-based Informed RRT*

The wrapping-based informed RRT* aims to further speed up the convergence of the informed RRT*. It works by first applying optimisation or refinement to each newly-found informed RRT* solution before the cost of such a solution is used to bound the search space in the subsequent search. The result is a reduction in the size of the resulting ellipsoid. This improves the rate of convergence of the informed RRT* towards the optimal solution. Kim and Song refer to their path-optimisation strategy as the *wrapping process* hence the name wrapping-based informed RRT*. In their context, wrapping refers to modifying an input path by shortening it in a manner that causes it to *wrap* around obstacles as illustrated in Figure 2.15. Starting with a path given by the sequence of n path nodes $\mathbf{Q} = \{\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_{n-1}\}$, the wrapping process discretises the solution path linearly, according to a user-defined resolution parameter. Once the path has been discretised, the wrapping process starts from the initial configuration and proceeds towards the goal configuration using the points on the discretised path in attempting to “push” intermediate nodes of the original path onto the obstacle. The exact process by which this “pushing” of the intermediate nodes is achieved is explained next with reference to Algorithm 15 and Figure 2.15.

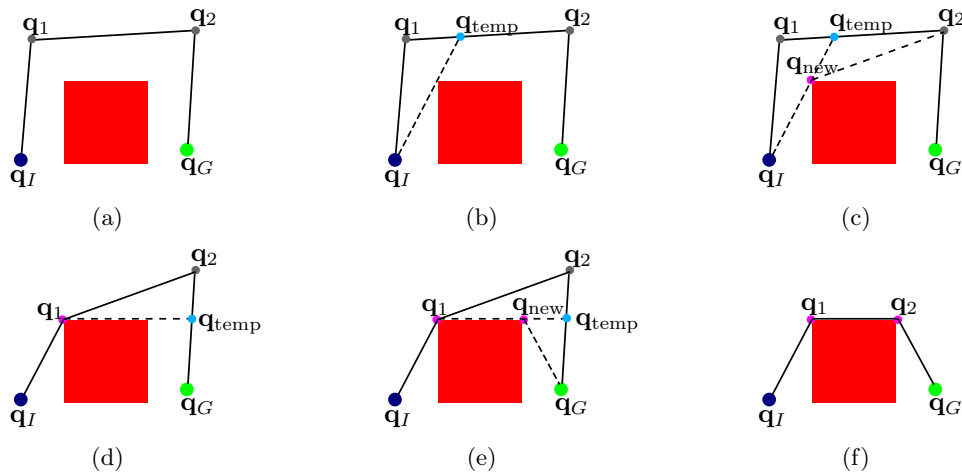


Figure 2.15: An illustration of the wrapping process used by the wrapping-based informed RRT*. Adapted from Kim and Song [98].

The algorithm begins from the initial configuration, whose index on the node path is $i = 0$. For each indexed node (i.e. for the i^{th} node, \mathbf{q}_i), the algorithm discretises the local path between \mathbf{q}_{i+1} and \mathbf{q}_{i+2} according to a discretisation parameter, D_1 . Then, starting from \mathbf{q}_{i+2} and moving backwards towards \mathbf{q}_{i+1} in steps of the first discretisation parameter, D_1 , the algorithm computes a temporary configuration, \mathbf{q}_{temp} , at each step (line 4), and attempts to connect configuration \mathbf{q}_i directly to \mathbf{q}_{temp} . On finding the first instance of \mathbf{q}_{temp} whose connection from \mathbf{q}_i is collision-free, the algorithm stops moving towards \mathbf{q}_{i+1} , essentially settling for that \mathbf{q}_{temp} (Figures 2.15(b) and 2.15(d)). The configuration \mathbf{q}_{i+1} is pruned from the path if it is possible to exactly connect \mathbf{q}_i to $\mathbf{q}_{temp} = \mathbf{q}_{i+2}$ (lines 6–9). Otherwise, starting from \mathbf{q}_i and moving forward towards \mathbf{q}_{temp} in steps of the second discretisation parameter, D_2 , the algorithm computes a new configuration, \mathbf{q}_{new} , at each step (line 12), and attempts to connect \mathbf{q}_{new} to \mathbf{q}_{i+2} . On finding the first instance of \mathbf{q}_{new} whose connection to \mathbf{q}_{i+2} is collision-free, the algorithm stops moving towards \mathbf{q}_{temp} , essentially settling for that \mathbf{q}_{new} (Figures 2.15(c) and 2.15(e)). This configuration is then used as a replacement configuration for \mathbf{q}_{i+1} . The local path from \mathbf{q}_i to \mathbf{q}_{new} is also used as a replacement local path for the local path between \mathbf{q}_i and \mathbf{q}_{i+1} , and the local path from \mathbf{q}_{new} to \mathbf{q}_{i+2} serves as a replacement for the local path between \mathbf{q}_{i+1} and \mathbf{q}_{i+2} (Figures 2.15(d) and 2.15(f), and lines 13–18 of the algorithm). This process repeats until the goal configuration is reached, at which point a modified path that wraps around the obstacle is obtained as shown in Figure 2.15(f).

Algorithm 15: wrap(node path \mathbf{Q})

```

1  $i \leftarrow 0$  ; // counter for traversing over path nodes.
2 while  $i < |\mathbf{Q}| - 2$  do
3   for  $k = 0$  to  $D_1 - 1$  do
4      $\mathbf{q}_{\text{temp}} \leftarrow \mathbf{q}_{i+2} + (\mathbf{q}_{i+1} - \mathbf{q}_{i+2}) \cdot \frac{k}{D_1}$  ; /* the  $k^{\text{th}}$  discrete configuration from  $\mathbf{q}_{i+2}$ 
        towards  $\mathbf{q}_{i+1}$  */
5     if  $\text{LPM}(\mathbf{q}_i, \mathbf{q}_{\text{temp}}) \in \mathcal{C}_{\text{free}}$  then
6       if  $k = 0$  then // Node  $\mathbf{q}_{i+1}$  bypassed, prune it
7         Prune configuration  $\mathbf{q}_{i+1}$  from path;
8          $i \leftarrow i - 1$ ;
9         break;
10    else
11      for  $m = 1$  to  $D_2 - 1$  do
12         $\mathbf{q}_{\text{new}} \leftarrow \mathbf{q}_i + (\mathbf{q}_{\text{temp}} - \mathbf{q}_i) \cdot \frac{m}{D_2}$  ; /* the  $m^{\text{th}}$  discrete configuration from  $\mathbf{q}_i$ 
            towards  $\mathbf{q}_{\text{temp}}$  */
13        if  $\text{LPM}(\mathbf{q}_{\text{new}}, \mathbf{q}_{i+2}) \in \mathcal{C}_{\text{free}}$  then
14           $\mathbf{q}_{i+1} \leftarrow \mathbf{q}_{\text{new}}$  ; /*  $\mathbf{q}_{\text{new}}$  is the replacement for  $\mathbf{q}_{i+1}$  that wraps onto
              obstacle. */
15          break;
16        if  $m == D_2 - 1$  then
17           $\mathbf{q}_{i+1} \leftarrow \mathbf{q}_{\text{temp}}$  ; /*  $\mathbf{q}_{\text{new}}$  is the replacement for  $\mathbf{q}_{i+1}$  that wraps onto
              obstacle. */
18          break;
19     $i \leftarrow i + 1$ ;

```

With the wrapping process in place, the wrapping-based informed RRT* algorithm works exactly the same way as the informed RRT* algorithm except that after each solution is found, it is processed by the wrap function, causing it to wrap around obstacles and consequently reducing its cost before it is used to bound the sampling region for subsequent iterations. This accelerates the rate of convergence towards the optimal solution. Pseudocode for the wrapping-based informed RRT* is given in Algorithm 16. The pseudocode is the same as that of the informed RRT*, save for the wrapping of each solution found (line 10)

Algorithm 16: wrappingBasedInformedRRT*($\mathbf{q}_I, \mathbf{q}_G$)

```

1  $V \leftarrow \{\mathbf{q}_I\}$ ;  $V \leftarrow \emptyset$ ;  $\mathcal{T} \leftarrow (V, E)$ ;  $\mathbf{Q}_{\text{sol}} \leftarrow \emptyset$ ;  $c_{\text{best}} \leftarrow \infty$ ;
2 for iteration  $\leftarrow 1$  to maxIterations do
3   if  $\mathbf{Q}_{\text{sol}} \neq \emptyset$  then
4      $\mathbf{q}_{\text{sampled}} \leftarrow \text{informedSample}(\mathbf{q}_I, \mathbf{q}_G, c_{\text{best}})$ ;
5      $\mathcal{T} \leftarrow \text{extendAndRewireRRT}^*(\mathcal{T}, \mathbf{q}_{\text{sampled}})$ ;
6   else
7      $\mathbf{q}_{\text{sampled}} \leftarrow \text{randomSample}()$ ;
8      $\mathcal{T} \leftarrow \text{extendAndRewireRRT}^*(\mathcal{T}, \mathbf{q}_{\text{sampled}})$ ;
9   if new solution found then
10     Apply the wrap algorithm to the new solution;
11      $\mathbf{Q}_{\text{sol}} \leftarrow \mathbf{Q}_{\text{sol}} \cup \text{new wrapped solution}$ ;
12      $c_{\text{best}} \leftarrow \text{cost of wrapped solution}$ ;

```

At the time of writing this thesis, the wrapping-based informed RRT* was the latest algorithm that builds on the informed RRT* with an aim of improving the rate of convergence RRT*-based

optimal path planners. It provides a technique that uses path optimisation within a sampling-based path planner to speed up its convergence to the optimal solution. While the technique presented did demonstrate the utility of using path optimisation within the informed RRT*, a number of issues can be pointed out with the particular path optimisation strategy used. Firstly, it does not incorporate robot kinematic and dynamic constraints – in the paper, a simple straight line LPM was considered. It is interesting to extend and demonstrate this idea for problems with motion constraints where a simple straight line LPM does not suffice – such as the problem considered by this thesis. Secondly, the path optimisation strategy used, i.e. *wrapping*, has not been compared with other path optimisation strategies such as gradient-based path optimisation. It is thus worthwhile to implement other path-optimisation algorithms and compare their effectiveness to that of the wrapping procedure when used in a similar way to accelerate the convergence of the informed RRT*. A review of existing path-optimisation techniques in literature is the subject of Section 2.8. Lastly, while the idea of using path optimisation within a sampling-based path planner was applied to the informed RRT* in Kim and Song’s work, it is interesting to consider the effect of using such a strategy within an anytime RRT path planner and comparing performance of such an algorithm to that applying path optimisation to the informed RRT*. With one of the objectives of this project being optimised path planning, the project will implement these three mentioned extensions to the wrapping-based informed RRT* as a way of furthering research efforts in optimal path planning using sampling-based path-planning algorithms.

At this point, single-query sampling-based optimal path planners have been considered. These algorithms solve the problem that this thesis aims to address in as far as path planning is concerned. After reviewing the existing algorithms of this category, modifications and extensions proposed by this thesis in furthering research on these algorithms have been outlined. The implementation and analysis of these proposed modifications and extensions is the subject of Chapter 5. Throughout our discussion on sampling-based path planners upto this point, the existence an algorithm known as the local path-planning method (LPM), that generates a feasible path between two configurations in \mathcal{C} , has been assumed. The review of existing local path-planning algorithms is the subject of Section 2.7. It is in that section that a suitable local path-planning method, among existing ones, will be selected for use in this project. Since this project is concerned with optimised path planning in static environments, the sampling-based path-planning algorithms that are of interest and relevance to the objectives of this project have been identified and reviewed. Other categories of sampling-based path planners include replanning and anytime replanning algorithms – these are useful for planning in dynamic environments. Due to the fact that these algorithms are not relevant to the objectives of the project, they are only discussed briefly in the following subsection for completeness sake.

2.6.5 Single-query Sampling-based Replanning

The path-planning techniques presented in the previous subsection do not take into account obstacles that unpredictably appear, move and disappear. Environments with such obstacles are said to be dynamic. A robot operating in such environments needs to be equipped with a planning algorithm capable of replanning, i.e. computing an alternative plan once an existing plan is invalidated by obstacle movements. As mentioned in the discussion of replanning in grid costmaps, replanning can take two forms. One form is that of planning anew whenever an existing plan has been invalidated – this is known as brute-force replanning. The other form is that of repairing the invalidated portions of the existing plan. A challenge with the first approach is limited planning time – there might not be enough time to plan from scratch every time changes in the environment are detected. A challenge with the second approach is that of keeping track of changes in the environment and propagating these changes to the rest of the plan. In this subsection, we briefly discuss existing replanning techniques in the context of sampling-based path planning. The brevity is due to these algorithms being outside the scope of the project, hence the discussion is only for completeness sake.

Just as optimal and anytime planning techniques in sampling-based path planners have been born out of ideas from assumptive grid-based path planning, so are replanning algorithms. One of the earliest sampling-based replanning algorithms is the ERRT algorithm developed by Bruce and Veloso in 2002 [103]. It caches path nodes for early solution paths and regrows the RRT

search tree from scratch when changes in the environment are detected. The cached nodes are then used to bias tree growth in the replanning phase. Later in 2006, Ferguson et al. introduced the DRRT algorithm [104], which is inspired by the grid-based replanning algorithm, D*. It plans from the goal to the start, just like the D*, to enable the reuse of the search tree. It thus served as an improvement to the ERRT, which did not reuse the same search tree. When changes in the environment are detected in DRRT, tree branches affected by the changes are pruned and deleted. Sampling is then biased towards these affected portions of the tree with the aim of growing new replacement branches. Zucker et al. then developed the MP-RRT [105] algorithm, which only prunes tree branches affected by environment changes, without deleting them. It essentially disconnects affected branches and actively attempts reconnecting them to the tree in future. While presenting an improvement to the DRRT by not deleting pruned branches, MP-RRT uses forward search instead of backward search. This attribute of the MP-RRT is considered as a retrogression from DRRT [106]. This is because as pointed out in the discussion on replanning in grid costmaps, for replanning to be efficient, it has to be done from the goal to the start since this allows the root of the search tree to remain the same for all replans, making it easier to reuse search efforts. It also makes it efficient to propagate changes detected by robot sensors, which usually have a short range and thus detect changes in the vicinity of the robot position which corresponds to the initial configuration. A common attribute among the replanning algorithms discussed so far is that they do not have any provable optimality guarantees. A replanning algorithm that was introduced with optimality guarantees is the RRT^X [106]. It has asymptotic optimality guarantees just like the RRT* algorithm discussed under optimal planners in the previous subsection and it can thus be referred to as an asymptotically optimal replanner. In contrast to its predecessors that prune and regrow some or all of the search graph, the RRT^X makes use of a rewiring process, similar to that of the RRT*, to remodel the tree around obstacles with the aim of repairing tree branches invalidated by changes in obstacle configurations and to update the costs of affected nodes. Similarly to optimal sampling-based path planning research that, after the introduction of optimal planners like the RRT*, strives towards accelerating the rate of convergence of asymptotically optimal sampling-based path planners, after the introduction of the RRT^X, sampling-based replanning research also focuses on the acceleration of the rate of convergence of asymptotically optimal replanners.

This subsection provided a brief overview of single-query sampling-based replanning algorithms. Although these algorithms are beyond the scope of the project, they have been reviewed for the sake of completeness and for highlighting their relation to single-query optimal sampling-based path planning, which is the focus of this project with regard to path planning. The next subsection concludes our review of sampling-based path-planning techniques.

2.6.6 Conclusion on Sampling-based Path Planning

This section reviewed path-planning algorithms that are based on the sampling-based path-planning approach. Similarly to grid-based path planners, this approach does away with the unrealistic assumption of perfect knowledge of obstacle shape and position that is associated with combinatorial path planners. However, the manner in which sampling-based path planning does away with this assumption is superior to that of grid-based path planning. Particularly, grid-based path planning discretises the configuration space while sampling-based path planning avoids this discretisation and instead directly samples the configuration space and uses a collision detector to enforce collision avoidance. The problem with discretising the configuration space is that it brings back the curse of dimensionality – by avoiding discretisation, sampling-based path planners overcome this curse. Moreover, both combinatorial and grid-based path planning do not incorporate vehicle motion constraints – hence failing to meet an important requirement for the path planning of this project. Sampling-based path planners provide an elegant and modular way for incorporating motion constraints, again having an edge over both combinatorial path planners. These advantages of sampling-based path planners led to their selection for use in this project.

With sampling-based path planning selected as the suitable approach for the optimised path planning problem addressed by this thesis, existing algorithms for addressing the optimised path planning problem through sampling-based path planning have been identified and discussed. Single-query sampling-based path planners were selected over their multi-query counterparts due to their relevance to the nature of the path planning task of the project – the task is to enable a single robot

to plan a path from an initial configuration to a goal configuration (i.e. single-query). Different types of single-query sampling-based path planners have been reviewed. These are feasible path planners, anytime path planners, optimal path planners and replanners. Replanners address the problem of planning in dynamic environments, which is outside the scope of the project – they were only reviewed for the completeness of our exploration of existing single-query path planners. The relevant single-query sampling-based path planners for the problem addressed by this project are feasible path planners, anytime path planners, optimal path planners and we now summarise our findings regarding them and how we aim to use each of them.

The basic RRT is a feasible path planner that is quick and efficient in finding a path, but does so without taking the cost of generated solutions into consideration. As such, it generates paths that are highly suboptimal. The anytime RRT is quick to find an initial, possibly highly suboptimal, path, early on, to ensure that a path can be available should planning time run out. It then keeps on searching for cheaper replacement solutions while planning time is available. The cheaper replacement solutions are found by focusing configuration space sampling using the cost of the current best solution, encouraging each search to find a better solution. The RRT* is an asymptotically optimal path planner that is guaranteed to find the optimal solution if given enough planning time, specifically with the number of iterations approaching infinity. The informed RRT* borrows the idea of focusing configuration space sampling using the cost of the current best solution from the anytime RRT to accelerate the rate of convergence of the RRT*. Furthermore, the wrapping-based informed RRT* accelerates the convergence of the informed RRT* by applying a path-optimisation step to each found solution to reduce its cost before it is used to focus configuration space sampling.

With the focus of the project also being path optimisation, a planner like the basic RRT which generates highly suboptimal paths is useful for testing the effectiveness of path-optimisation algorithms once they have been developed. The basic RRT will thus be developed and adapted for continuous-curvature path planning. With regard to the anytime RRT, there has not been reported work on the application of a path-optimisation step to accelerate its rate of convergence in a similar way a path-optimisation step is used to accelerate the rate of convergence of the informed RRT* in the wrapping-based informed RRT*. This improvement of the anytime RRT will be developed and adapted for continuous-curvature path planning in this project. Moreover, other path-optimisation techniques will be considered as replacements for the wrapping process in our accelerated version of the anytime RRT and their effectiveness compared to the wrapping process. The informed RRT* will also be developed and adapted for continuous-curvature path planning. Other path-optimisation techniques, apart from the wrapping process will be considered for the acceleration of the informed RRT* and their effectiveness compared to that of the wrapping process.

The proposed adaptations and extensions are enumerated below for convenience:

1. Adaptation of the basic RRT for continuous-curvature path planning by developing and using a continuous-curvature local path planner.
2. Extending the anytime RRT algorithm in a similar way as the extension of the informed RRT* to the wrapping-based informed RRT*.
3. Adaptation of the wrapping-based informed RRT* for continuous-curvature path planning by using the local path planner mentioned in the first point.
4. Comparison of the extended anytime RRT algorithm to the adapted wrapping-based informed RRT*.
5. Identifying and implementing other path-optimisation algorithms that can be used in place of the wrapping process, and comparing the effectiveness of these algorithms to that of the wrapping process.
6. Application of other optimisation strategies as replacement for the wrapping process accelerated anytime RRT and the wrapping-based informed RRT*.

The proposed adaptations and extensions mention two algorithms which have not been reviewed yet, namely local path planning and path optimisation. Review of literature for existing methods

for these algorithms is the subject of the next two sections, with Section 2.7 discussing local path planning and Section 2.8 focusing on path optimisation literature.

2.7 Local Path Planning

In the preceding discussions on sampling-based path planning, referred to as a local path-planning method (LPM) that connects two given configurations with a feasible path for a given vehicle model. The various methods used to achieve this in the context of vehicles with a constrained turning rate are a subject of this section.

There are two widely-studied models for vehicles with a constrained turning rate, namely *Dubins car* [107] and the *Reeds-Shepp car* [108]. While the Dubins and Reeds-Shepp models provide intuitive solutions for local path planning that have been applied extensively for local path planning in practical systems, they both suffer from a drawback of failing to satisfy the constraint on maximum curvature derivative associated with vehicles targeted by this thesis. As such, they are not sufficient for a path planner that meets the requirements of the thesis. Newer methods known as continuous-curvature path planners strive to overcome this shortcoming by improving on both Dubins and Reeds-Shepp models so as to account for the bound on the rate of change of curvature. The Dubins local path planner plans local paths for a vehicle that is constrained to only move in the forward direction with a constant velocity, and has a minimum turning radius, r_{\min} [109]. A Dubins vehicle model can apply three steering commands namely: turn left at radius r_{\min} , turn right at radius r_{\min} or move straight. The result of limiting steering commands to these three is that the model assumes an instantaneous change from moving straight to turning at radius r_{\min} and vice-versa, disregarding the existence of an upper bound on the rate of change of curvature. The Reeds-Shepp local path planner on the other hand plans local paths for a vehicle of the Dubins type, but which is also allowed to move backwards with constant velocity [25, 108]. Continuous-curvature (CC) local path planners [11, 110–112] extend both the Dubins and the Reeds-Shepp local path planners by allowing transition manoeuvres between straight-line and turning motion or between turns of different directions. Such transition manoeuvres respect the existence of the upper bound on the rate of change of curvature. As a result, the development of Dubins and Reeds-Shepp local path planners are used to develop the continuous-curvature local path planners. For this reason, our review of local path-planning techniques starts with the Dubins LPM, proceeds to Reeds-Shepp LPM and finally ends with continuous-curvature local path planning.

2.7.1 Dubins Local Path-planning Method

The Dubins car [107] is a vehicle model that is constrained to move forward (never backward) with constant velocity and can apply one of three steering commands, namely: turn left along an arc of radius r_{\min} , turn right along an arc of radius r_{\min} or move straight. Dubins showed that for such a car, given the initial and goal configurations, the shortest path consists of exactly three tangentially connected path segments which are either circular arcs of minimum turning radius or straight line segments. He proved that there are exactly 6 contenders for the shortest path. He classified these contenders as $L_t S_u L_v$, $R_t S_u R_v$, $L_t S_u R_v$, $R_t S_u L_v$, $L_t R_u L_v$ and $R_t L_u R_v$. In this notation L represents a left turn along an arc of radius r_{\min} , R denotes a right turn along an arc of radius r_{\min} , and S denotes a straight-line motion. The subscripts t , u and v specify the length of each path segment. Here left and right mean anticlockwise and clockwise respectively around a circle of radius r_{\min} , i.e. a tightest possible circle for the vehicle, where the circular arc is restricted to be less than a full circle. He further simplified the notation to the form CCC or CSC where C denotes a circular arc of radius r_{\min} and S denotes a straight line segment. With this generalisation, L and R in the original notation can be replaced with C , making LRL and RLR paths generally CCC paths and LSL , LSR , RSR and RSL paths generally CSC paths. Figure 2.16 shows example Dubins paths.

The Dubins local path-planning approach is highly intuitive and has been applied successfully in numerous of works in the literature, some of which have been included in the bibliography [113–118]. However, it has a number of drawbacks. Firstly, since it considers forward motion only, the car can easily get trapped in case of narrow cul-de-sacs. Figure 2.17 shows such a scenario. Secondly, the Dubin's model has a discontinuous-curvature profile with discontinuities occurring

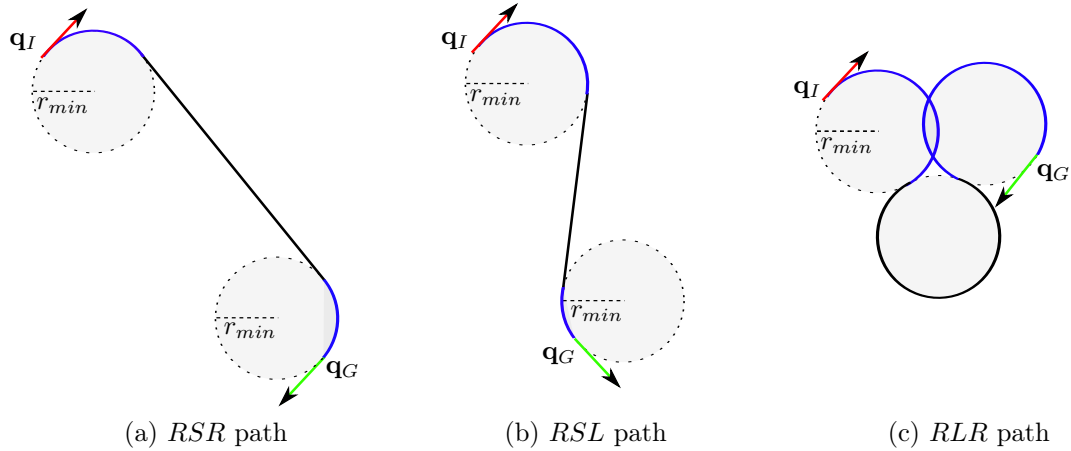


Figure 2.16: Example Dubins paths. The initial and goal configurations are shown by red and green arrows respectively. The first and last path segments are coloured blue and the middle one is coloured black. The dotted circles are the Dubins turning circles.

at the transitions between line segments and arcs and between arcs with opposite directions of rotation [11]. Thus for a vehicle with a bounded curvature derivative to accurately track such a path without stopping, it would have to change its front wheels' orientation instantaneously (which is impossible); otherwise it would have to stop at each curvature discontinuity and reorient its front wheels before proceeding. This is not desirable in most driving situations [119]. The Reed-Shepp LPM that is the subject of the following subsection, addresses the first issue with the Dubins LPM. The second issue is addressed by continuous-curvature path-planning techniques of Section 2.7.3 that follows next.

2.7.2 Reeds-Shepp Local Path-planning Method

Reeds and Shepp [108] extended the Dubins car of the preceding subsection by allowing it to move in reverse, resulting in the Reeds-Shepp car. They used the same word notation as Dubins, but in addition, they introduced a sign superscript to specify the direction of motion along the path segment, thus leading to the notation $L_t^\pm S_u^\pm L_v^\pm$, $R_t^\pm S_u^\pm R_v^\pm$, $L_t^\pm S_u^\pm R_v^\pm$, $R_t^\pm S_u^\pm L_v^\pm$, $L_t^\pm R_u^\pm L_v^\pm$, and $R_t^\pm R_u^\pm R_v^\pm$ or alternatively, $C_t^\pm S_u^\pm C_v^\pm$ or $C_t^\pm C_u^\pm C_v^\pm$. Here the plus sign implies forward movement, while the negative sign is equivalent to the reverse gear and t , u , and v are the respective lengths for each path segment. They also introduced a more compact representation that eliminates the use of the \pm symbol and instead simply use the vertical line symbol, $|$, to indicate reverse motion. The authors derived 48 contenders for the shortest path (compared to Dubins' 6), and later Sussmann [120] completed the derivation, lowering this number to 46.

An interesting realisation of the Reeds-Shepp local path planner is that it sometimes shortens Dubins paths. This is demonstrated through the “reverse-in-place” problem illustrated in Figure 2.18, where the challenge is to find the shortest path that returns a vehicle to its initial position but in a goal orientation that is exactly opposite to the initial orientation. This can be seen from the contending Dubins and Reeds-Shepp paths in Figure 2.18. It is clear from the figure that

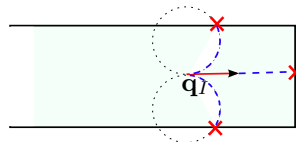


Figure 2.17: An illustration of a Dubins car trapped in a narrow cul-de-sac due to its inability to execute reverse motions. Attempts to turn along arcs of minimum turning radius as well as an attempt to move straight-forward would result in collisions as indicated by the red crosses.

the path in (d) is shorter than all three Dubin's paths ((a), (b), (c)). Moreover, as stated earlier, reverse capability is sometimes needed to avoid the vehicle being trapped in narrow passages.

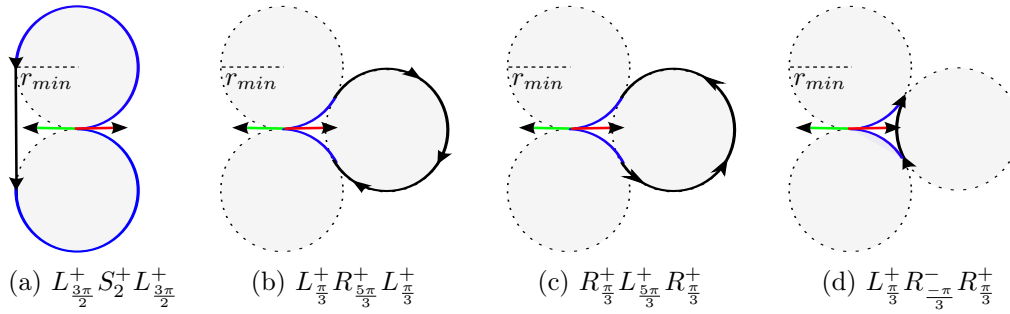


Figure 2.18: The shortest Reeds-Shepp path is sometimes shorter than all contending Dubin's paths. In this illustration of the reverse-in-place problem, executing reverse motions (subfigure (d)) results in a shorter path than all contending Dubins paths (subfigures (a), (b) and (c)).

While Reeds and Shepp made great progress by allowing the vehicle to move backwards, their model still suffers from the discontinuity in curvature between path segments just like the Dubins model. *Continuous-curvature local path planning* is thus an active subject in autonomous vehicles research. The following subsection reviews existing methods that improve Dubins and Reeds-Shepp models to ensure curvature continuity in the planned paths.

2.7.3 Continuous-curvature Local Path-planning Method

Continuous-curvature local path planning improves Dubins and Reeds-Shepp local path planners by introducing transition manoeuvres between straight-line motion and a turn as well as between two turns of opposite directions. These transition manoeuvres account for the bounded rate of change of curvature resulting from the upper bound on angular acceleration that a vehicle is capable of.

Key pioneering works in continuous-curvature path planning were presented by Boissonnat, Kostov, Sussmann, Scheuer and Fraichard [11, 110–112, 121–123]. Particularly, Fraichard and Scheuer have contributed significantly to this subject and have published a number of works taking into account constraints on both the curvature and sharpness of the planned path using clothoids [11, 110–112].

By definition, a clothoid is a curve whose curvature increases linearly with its length according to Equation 2.15 that gives the instantaneous curvature at a point at distance s along the clothoid from its starting point as:

$$\kappa(s) = \kappa_0 + \alpha s, \quad (2.15)$$

where κ_0 is the curvature at the beginning of the curve, α is its sharpness (also known as the curvature derivative) and s is the path length parameter (note that if we assume a straight line initial motion, i.e. $\kappa_0 = 0$, the equation reduces to $\kappa(s) = \alpha s$). This property makes the clothoid useful as a transition curve on a path between a straight line and a circular arc because the curvature is continuous and a vehicle following the curve at constant speed will have a constant rate of angular acceleration [12]. Furthermore, clothoids are widely used in civil engineering in the design of road networks, making it more suitable for path planning in on-road autonomous vehicles. Wilde [12] notes that the paths planned by Fraichard and Scheuer always turn the steering wheel at the maximum allowable rate until the minimum turning radius is reached and thus all turns are performed as tight as possible and thus may result in discomfort. He proposes an approach that uses a minimal amount of steering (i.e. clothoids of minimal sharpness) to reach the desired terminal position similar to the way a human would drive. Other contributions on the use of clothoids for trajectory generation can be found in the work presented by Bakolas, Brezak, Girbes, Meidenbauer and Henrie [124–128] while the geometry of clothoids is well described by Meek [166]. Some researchers have explored the use of other curves such as Fermat's spiral [131] and Bezier

curves [130], but the clothoid remains superior due to its linear curvature profile. For completeness, the geometry of clothoids and their application to trajectory generation will be introduced in the discussion of local path planner design and implementation (Chapter 4) with reference to the above mentioned literature. The next subsection concludes our local path planning literature review by summarising our findings.

2.7.4 Conclusion on Local Path-planning

Three classes of local path planners for vehicles with a constrained turning rate have been considered, with each subsequent planner improving on deficiencies of the previous one. Two issues with Dubins LPM have been mentioned. The first one is the fact that it only allows forward motion and as such, it can easily get trapped in narrow cul-de-sacs. The second issue is that the paths it produces do not satisfy vehicle motion constraints, particularly that they are not curvature continuous. On the other hand, only one issue has been mentioned with Reeds-Shepp LPM, and that is the discontinuity in the curvature profile of the paths it produces. The second issue with Dubins LPM is the same as the issue with Reeds-Shepp LPM. This issue can be addressed by incorporating continuous-curvature transitions between the Dubins or Reeds-Shepp path's segments as discussed in the preceding section. With this issue addressed, no issue remains with Reeds-Shepp LPM while one issue remains in Dubins LPM. Naturally, this evaluation gives a continuous-curvature adaptation of Reeds-Shepp LPM an edge over a continuous-curvature adaptation of Dubins LPM. However, a general advantage of Dubins LPM over Reeds-Shepp LPM is its simplicity. As a result, in situations where this advantage can be exploited without issues, the use of Dubins LPM can be beneficial both in terms of the simplicity of the development of the system and in terms of its efficiency. With that said, it is important to note that the remaining issue of Dubins LPM, i.e. the risk of getting trapped in narrow cul-de-sacs, cannot manifest when the LPM is used within a global path planning scheme such as the one to be implemented in this project, as long as the initial configuration is not in a narrow cul-de-sac. This is because the global path planner ensures the end-to-end feasibility of the planned path before it is executed. As a result, as long as the initial configuration has enough clearance on all sides, a continuous-curvature adaptation of Dubins LPM is suitable for the local path planning requirement of this project. Due to this reason, a continuous-curvature adaptation of Dubins LPM has been selected to address the local path planning aspect of this project. This adaptation is the subject of Chapter 4.

Local path planning was one of two algorithms mentioned in the adaptations and extensions on existing sampling-based path planners proposed by this project. With local path planning discussed and a suitable local path-planning method selected, in the next section we proceed to review path optimisation literature with the goal of selecting suitable path-optimisation algorithms.

2.8 Path Optimisation

Optimisation is the minimisation or maximisation of a cost function (also known as the objective function) subject to constraints on its variables [132]. The objective is a quantitative measure of the performance of the system under consideration and it depends on certain characteristics of the system known as variables. These variables are often constrained, i.e. they are not allowed to take any arbitrary value. In the optimisation problem considered by this thesis, we are interested in the minimisation of the length of a solution path found by a sampling-based path planner in the configuration space, \mathcal{C} . The objective is thus path length and the variables are configurations that may be part of the solution path. These configurations are constrained – configurations lying in the obstacle region, \mathcal{C}_{obs} , are prohibited from being part of the solution path.

As stated in Subsection 2.6.2, solutions returned by algorithms based on the RRT are almost-surely suboptimal. Those based on the RRT* are asymptotically optimal – they are guaranteed to converge almost-surely to the optimal solution as the number of iterations approaches infinity. It is therefore beneficial to feed the paths produced by both RRT and RRT* algorithms to a path optimiser whose task is to minimise the cost of the resulting solutions – in our case path length. In the RRT case, path optimisation is the only hope if optimality is to be attained as on its own, an RRT algorithm is almost-surely suboptimal. In the case of RRT* algorithms, a path optimiser helps to accelerate the rate of convergence of the algorithm towards the optimal

solution when the cost of the optimised solution is used to iteratively focus the algorithm as in the wrapping-based informed RRT*. In Section 2.6.6, it was proposed to adapt the wrapping process used by the wrapping-based informed RRT* for continuous-curvature path optimisation and to apply this wrapping strategy to the anytime RRT algorithm and compare the performance of the resulting algorithm to that of the adapted wrapping-based informed RRT*. It was also proposed to extend this idea by considering other optimisation algorithms that can be used in place of the adapted wrapping process and to evaluate the effectiveness of such strategies compared to the wrapping process. This section reviews existing path-optimisation algorithms. The next two subsections discuss two classes of path-optimisation algorithms found in existing literature. These are shortcut-based path optimisation and gradient-based path optimisation. They are respectively treated in Subsections 2.8.1 and 2.8.2.

2.8.1 Shortcut-based Path Optimisation

Shortcut-based path optimisation is considered a simple technique for decreasing the path length of a node path [133]. A node path is a path represented by a sequence of nodes $\mathbf{Q} = \{\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_{n-1}\}$, where each node pair $(\mathbf{q}_i, \mathbf{q}_{i+1})$ is connected by a collision-free local path. Shortcut-based path optimisation essentially works by bypassing node(s) on the path as illustrated in Figure 2.19, if it possible to do so. A node, \mathbf{q}_k , on the path with k in the range $i < k < j$ is considered redundant if there exists a local path from \mathbf{q}_i to \mathbf{q}_j ($j > i$) that is collision-free. Figure 2.19(a) shows a case of a single redundant node, while Figure 2.19(b) shows a case of multiple redundant nodes. There is generally no limit to the number of nodes that can be bypassed, as long as a feasible shortcut path exists. Different shortcut-based path-optimisation algorithms differ in how they select the node pair $(\mathbf{q}_i, \mathbf{q}_j)$ between which a shortcut is to be attempted. In the next three subsections, we consider three existing shortcut-based path-optimisation algorithms. These are, respectively, the *path pruning method*, the *random shortcut method* and the *wrapping process*.

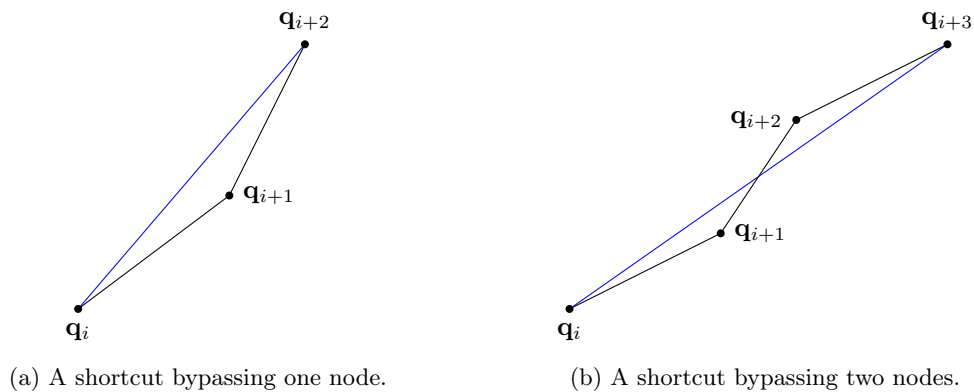


Figure 2.19: An illustration of the application of shortcut-based path optimisation for shortening the length of a node path.

2.8.1.1 Path Pruning

The simplest form of shortcut-based path optimisation is known as *path pruning* [133]. Under this technique, a node, \mathbf{q}_{i+1} , on the path is considered redundant if there exists a local path from \mathbf{q}_i to \mathbf{q}_{i+2} that is collision-free. In essence, it only considers shortcuts that bypass one node (Figure 2.19(a)). Pseudocode for the path pruning algorithm is given in Algorithm 17 and illustrated in Figure 2.20. The algorithm starts from the first node of the node path, \mathbf{q}_0 and attempts to bypass the second node \mathbf{q}_1 . If this operation is successful, the redundant node \mathbf{q}_1 is removed from the node path and \mathbf{q}_0 is connected directly to \mathbf{q}_2 . The example given in Figure 2.20 demonstrates a scenario where this operation was successful. In successive iterations, the algorithm

proceeds along the new shortened node path, attempting to remove more redundant nodes if any exists.

Algorithm 17: `prunePath(node path Q)`

```

1  $i \leftarrow 0$  ; // Start from the first node of the node path,  $q_0$ 
2 while  $i < |Q| - 2$  do
3   if collisionFree(LPM( $q_i, q_{i+2}$ )) then
4      $Q \leftarrow Q \setminus q_{i+1}$  ; // Remove redundant node from node path
5     if  $i > 0$  then
6        $i \leftarrow i - 1$ 
7     else
8        $i \leftarrow i + 1$ 
9 return  $Q$ 

```

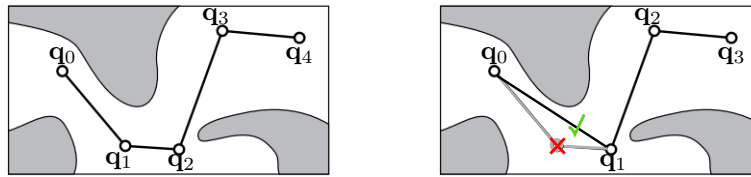


Figure 2.20: Original path(left) and one node pruned from path (right)

The pruning method discussed above only uses the nodes of a node path to attempt shortcuts that shorten the path. Consequently, the ability of this method to reduce the length of a given node path depends on the visibility properties of its nodes. In particular, a shortcut is only attempted if it is possible to connect two non-consecutive path nodes. With no non-consecutive nodes connectible, although it may be possible to otherwise shorten the path, the algorithm will fail. The random shortcut method that is discussed next overcomes this shortcoming of path pruning.

2.8.1.2 Random Shortcut Method

The *random shortcut method*, whose pseudocode is given in Algorithm 18 and illustrated in Figure 2.21, overcomes the shortcoming of the pruning method of relying on the visibility of non-consecutive path nodes. It achieves this improved performance by not only considering the nodes of node path for path shortening, but discretising the node path according to a discretisation step parameter (line 3 of the algorithm and Figure 2.21(b)) and then using the configurations produced by this discretisation to attempt shortcuts. It is an iterative method and at each iteration it picks two configurations, q_a and q_b at random (line 4 of the algorithm). These picked configurations split the path into three parts: The first part is that part between the initial configuration, q_I , and q_a (line 5 of the algorithm), the second one is that between q_a and q_b (line 6 of the algorithm) and the third one is the one between q_b and the goal configuration, q_G (line 7 of the algorithm). With the path split into these three parts, the algorithm checks if it is possible to replace the middle part (i.e. the one between q_a and q_b) with a shorter feasible and collision-free path (Figures 2.21 (c), (d), (e) and line 8 of the algorithm). If this operation succeeds, the path is modified (line 9 of the algorithm) and the same process repeats in subsequent iterations until the algorithm terminates. Figure 2.21(f) shows an example final path resulting from the application of random shortcut method to the initial node path given in Figure 2.21(a).

By not only considering the nodes of the node path, but also intermediary configurations determined by the discretisation step, the random shortcut method outperforms path pruning in producing shorter paths. It does so at the expense of increased computation time. While the pruning method is deterministic and conclusive, i.e. it runs until all possible shortcuts between

Algorithm 18: randomShortcut(discretised node path \mathbf{Q})

```

1  $K \leftarrow$  total number of iterations;
2 for iteration = 1 to  $K$  do
3    $n \leftarrow$  number of configurations on discretised node path,  $\mathbf{Q}$ ;
4    $(a, b) \leftarrow a_{\text{rand}}, b_{\text{rand}} : 0 \leq a_{\text{rand}} + 1 < b_{\text{rand}} < n$  ; /* pick two random configurations on
      the path that split the path into three parts. */
5    $\mathbf{Q}' \leftarrow \mathbf{q}_0, \dots, \mathbf{q}_{a-1}$  ; // the first part of the split path.
6    $\mathbf{Q}'' \leftarrow \mathbf{q}_a, \dots, \mathbf{q}_b$  ; // the second part of the split path.
7    $\mathbf{Q}''' \leftarrow \mathbf{q}_{b+1}, \dots, \mathbf{q}_{n-1}$  ; // the third part of the split path.
8   if collisionFree(LPM( $\mathbf{q}_a, \mathbf{q}_b$ )) then
9      $\mathbf{Q} \leftarrow \mathbf{Q}' \cup \text{LPM}(\mathbf{q}_a, \mathbf{q}_b) \cup \mathbf{Q}'''$  ; // replace middle part of path.
10 return  $\mathbf{Q}$ ;
    
```

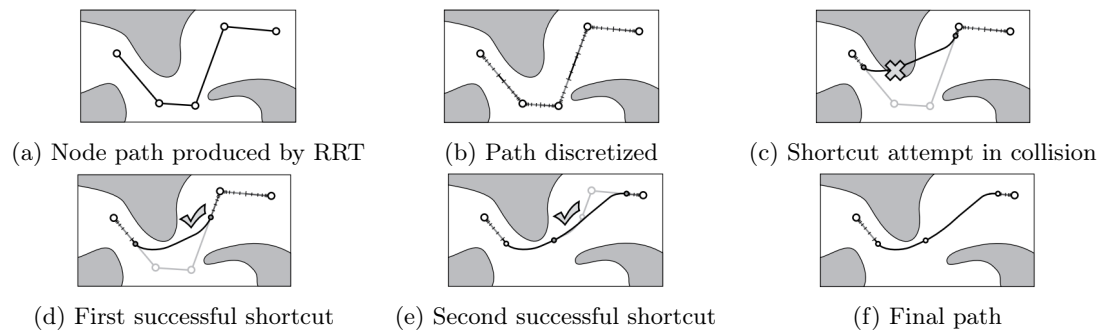


Figure 2.21: An illustration of the application of the random shortcut method to a node path. Subfigure (a) shows the initial node path. The path is then discretised in Subfigure (b) and in Subfigure (c), the random configurations \mathbf{q}_a and \mathbf{q}_b are selected and a shortcut is attempted. This attempt results in a collision and it is thus discarded. Subfigures (d) and (e) show situations where the shortcut attempt succeeds and Subfigure (f) shows the resulting path.

nodes are exhausted, the random shortcut method is probabilistic. Other variants of the random shortcut can be found in a number of works in literature [134, 135]. Deterministic versions of the shortcut method where the configuration pair $(\mathbf{q}_a, \mathbf{q}_b)$ is selected deterministically also exist.

The next subsection discusses the last among considered shortcut-based path-optimisation algorithms, the wrapping process.

2.8.1.3 Wrapping Process

The wrapping process described in Algorithm 16 and illustrated in Figure 2.15 of Subsection 2.6.4 employs a path optimisation strategy that is somewhat similar to a deterministic shortcut method, except that in addition to applying a shortcut between selected configurations, the wrapping process also moves configurations such that the resulting path tightly wraps around obstacles. This algorithm was explained in detail in the discussion of the wrapping-based informed RRT*. It is not discussed here to avoid repetition.

The next subsection provides a conclusion on the considered shortcut-based path-optimisation techniques.

2.8.1.4 Conclusion on Shortcut-based Path Optimisation

We have now reviewed path-optimisation techniques that shorten the length of an initial node path by making use of shortcuts. The simplest within this class of path-optimisation algorithms is the path pruning method. However, this method relies on the visibility properties of the nodes of the input node path. Without visibility between non-consecutive nodes, the algorithm fails. An improvement to this algorithm which overcomes the aforementioned shortcoming is the random

shortcut method. However, it offers this improvement at the cost of increased computation time. The wrapping process is a deterministic method that improves on the shortcut method by forcing the resulting paths to wrap around obstacles. Similarly to the random shortcut method, the wrapping process proved path length reduction at the cost of increased computation time. To limit computation time while still preserving the ability to produce shorter paths, the discretisation parameters used by both the random shortcut algorithm and the wrapping process have to be carefully selected.

With the main aim of the project being to investigate the effectiveness of path-optimisation algorithms in accelerating the rate of convergence of the path planners that were selected in Section 2.6.6, it has been decided to develop all the considered shortcut-based path-optimisation algorithms and to incorporate all of them into the selected path planners for the acceleration of the rate of convergence. This will help us to measure the effectiveness of each path-optimisation technique in accelerating the rate of convergence, relative to the effectiveness of its contending counterparts.

It is important to reiterate that all the considered path optimisers will be adapted for continuous-curvature path optimisation, i.e. so that they optimise an input continuous-curvature path in a manner that preserves its curvature continuity. This is to ensure that the produced paths can be accurately executed by vehicles with constrained kinematics. The importance of developing path optimisers that optimise paths while preserving their feasibility for vehicles with motion constraints is well stated by Geraerts et al. and Lavalle. In their presentation of the path pruning method and the random shortcut method, Geraerts et al. concede that while their versions of these path-optimisation techniques were shown experimentally to be effective in their work, the techniques were only applied to holonomic robots, i.e. ones with no motion constraints. They state that an interesting extension to their work would be adapting the algorithms for robots with motion constraints. Their admission is consistent with LaValle's view that in the absence of motion constraints, it is much simpler to shorten paths and that in the most general setting, it is very difficult to improve paths [25, p. 855]. This interesting extension is one of the focus areas of this project. It is not only applied to the path pruning and the random shortcut algorithms, but to all other path-optimisation techniques that will be developed, including the wrapping process and gradient-based path optimisation.

The next section considers algorithms that reduce the length of an initial node path by using gradient information, given a cost function.

2.8.2 Gradient-based Path Optimisation

Gradient-based path optimisation falls within numerical optimisation methods [136] and makes use of nonlinear programming techniques to minimise a cost function that measures the goodness of a path [25]. This is opposed to the shortcut-based path-optimisation techniques of the previous subsection, which do not explicitly minimise a cost function, but rather rely on iteratively attempting shortcuts that can possibly reduce the path length. As such, gradient-based path optimisation has clear termination conditions that are based on the numerical convergence of the cost of the path to the optimal solution as opposed to the shortcut-based optimisation approach in which termination is either based on the exhaustion of available shortcuts (as in the pruning method) or on a preset user-specified number of iterations (as in the random shortcut method). This section reviews existing techniques for gradient-based path optimisation.

One of the earliest and popular gradient-based path-optimisation algorithms is the CHOMP (short for covariant Hamiltonian optimisation for motion planning) algorithm introduced by Ratliff et al. [137, 138]. This algorithm uses covariant gradient techniques to improve the quality of an initial path. A unique property of CHOMP that separates it from other path-optimisation algorithms is that it can be used to both generate and optimise paths. It achieves this property by dropping the requirement for the initial path to be collision-free, which is common with other path-optimisation algorithms. Instead, the CHOMP algorithm can begin with a naive initial guess such as a straight line connecting the initial configuration to the goal and then transform this initial guess into an optimised feasible collision-free solution. In this manner, it can be used without using a separate path planner to generate the initial guess. The algorithm makes use of a covariant gradient update rule to minimise a cost function that is a combination of path smoothness and ob-

stacle clearance. The covariant gradient update rule is used so as to make sure that the algorithm converges quickly to a locally optimal solution [137]. A number of path-optimisation algorithms have been introduced as extensions of CHOMP, most notably STOMP (stochastic trajectory optimisation for motion planning) introduced by Kalakrishnan et al. [139] and ITOMP (iterative trajectory optimisation for motion planning) introduced by Park et al. [140]. STOMP builds on CHOMP, combining the advantages of CHOMP with the robustness of stochastic approaches. In particular, by employing a stochastic approach, it is able to handle cost functions for which gradient information is not available. It works by generating noisy paths as a means to explore the space around the initial, possibly infeasible, path. In each iteration of the algorithm, a series of noisy paths are generated and the costs of these generated paths are computed and used to update the current solution. The stochasticity of STOMP enables it to overcome the phenomenon getting stuck in a local minima which is possible in non-stochastic methods. ITOMP is an incremental path-optimisation algorithm for real-time replanning in dynamic environments [140]. To handle dynamic obstacles, it estimates the trajectories of moving obstacles and based on this predicted obstacle motion, a conservative bound on their position is computed. The path from the initial to the goal configuration is then computed by solving an optimisation problem that avoids collisions with obstacles while satisfying path smoothness constraints. Since in dynamic environments, the obstacle positions may change after optimisation has completed and while the optimised path is being executed, ITOMP alternates between path optimisation and path execution. Thus instead of solving the path optimisation problem completely, a time budget is assigned for path optimisation. The path optimiser is interrupted when this time runs out and the robot executes the current optimised path over a short time interval. As such, at any point in time, the current path may be suboptimal, i.e. its cost may not be minimised and may not be entirely collision-free. This process repeats until the robot reaches the goal configuration. A drawback of the CHOMP algorithm is that it is significantly complex algorithmically [141]. It also requires a pre-processing step of the robot and its environment model to make it simpler since it handles collision avoidance by inequality constraints sampled at many points along the path [136, 141]. A more recent algorithm has been proposed by Campana et al. [136]. It directly optimises path length and provides a way of handling collision constraints by adding them to the optimisation [141], without requiring any geometry pre-processing or offline optimisation to counterbalance costly distance computations [136]. With this project aiming at applying path optimisation to reduce path length of an initial solution, this algorithm yields itself versatile for our purpose. While Campana et al.'s algorithm offers a significant contribution for efficient path optimisation, it presently does not incorporate motion constraints. In its present state, it optimises paths that are composed of a concatenation of straight lines to produce an optimised path that is also a concatenation of straight lines. To be used in this project, the algorithm would have to be adapted for continuous path optimisation. In consistency with the views of Geraerts et al. and LaValle that were mentioned in our concluding remarks on shortcut-based path optimisation, it is interesting to extend path-optimisation algorithms to vehicles with motion constraints. Due to the relevance of Campana et al.'s algorithm to the path optimisation objective of this project, we now describe it in detail, with reference to their paper [136]. Prior to this description, it is important to introduce notation for path representation.

In this thesis, we consider a path with n nodes to be given by $\mathbf{Q} = \{\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_{n-1}\}$, where \mathbf{q}_i is the i^{th} configuration on the path, and \mathbf{q}_0 and \mathbf{q}_{n-1} respectively correspond the initial and goal configurations. In the following discussion of Campana's algorithm, we adapt their notation to conform with this notation.

The algorithm works as a classical linearly constrained quadratic program (LCQP). It reduces the length of a node path, represented as a cost function and avoids collisions through the use of linearised constraints. Since the objective is to minimise path length and the optimisation variables are the path nodes, it is necessary to express the length of the path in terms of the path nodes and to derive the path gradient from that expression. In their work, for which it is important to remark that it optimises paths for a manipulator, the length of the straight interpolation between two configurations, \mathbf{q}_i and \mathbf{q}_{i+1} is computed as:

$$\|\mathbf{q}_{i+1} - \mathbf{q}_i\|_W \triangleq \sqrt{(\mathbf{q}_{i+1} - \mathbf{q}_i)^T W^2 (\mathbf{q}_{i+1} - \mathbf{q}_i)}, \quad (2.16)$$

with this length being in the joint space, not in the Euclidean space. The length metric, W ,

represents the Mahalanobis distance which weighs joints. Since q_0 and q_{n-1} are fixed, with straight-line interpolations between consecutive configurations, the length of the node path is given as:

$$L(\mathbf{Q}) = \sum_{k=1}^{n-2} \|\mathbf{q}_k - \mathbf{q}_{k+1}\|_W^2, \quad (2.17)$$

where \mathbf{q}_k is the configuration of the k^{th} node on the node path and \mathbf{Q} is, again, the set of path configurations. The length of the node path, given by Equation 2.17, represents the cost that we wish to optimise. It is conditioned for path optimisation to:

$$L(\mathbf{Q}) = \frac{1}{2} \sum_{k=1}^{n-2} \lambda_{k-1} \|\mathbf{q}_k - \mathbf{q}_{k+1}\|_W^2, \quad (2.18)$$

where the coefficients, λ_{k-1} , are used to keep the ratio between the lengths of path segments of the optimised path the same as at the initial path. If these coefficients are not used, the optimised path has its path nodes located such that they are equidistant from each other – an irrelevant attribute, since some parts of the configuration space may be highly cluttered with obstacles while others are not. After differentiating the conditioned path cost of Equation 5.6 with respect to the path nodes, the gradient is given as follows:

$$\nabla L(\mathbf{Q}) = ((\lambda_k(\mathbf{q}_{k+1} - \mathbf{q}_k)^T - \lambda_{k+1}(\mathbf{q}_{k+2} - \mathbf{q}_{k+1})^T)W^2), \quad k \in 0, \dots, n-3. \quad (2.19)$$

The gradient expression is then used to form a second-order update rule given as follows:

$$\mathbf{Q}_{i+1} \leftarrow \mathbf{Q}_i - \alpha_i H^{-1} \nabla L(\mathbf{Q}_i)^T, \quad (2.20)$$

where \mathbf{Q}_{i+1} denotes the new iterate, \mathbf{Q}_i is the previous iterate, α_i is the step length parameter or learning rate, and H is the Hessian and can be obtained from the gradient expression (Equation 2.19). This second-order update rule offers an unconstrained solution to the optimisation problem. Particularly, with $\alpha_i = 1$, the algorithm finds the unconstrained minimum in one step. This minimum corresponds to the case where all the path nodes are aligned on a straight line, starting from the initial configuration and ending at the goal configuration. However, this solution contains collisions – if it did not, the path planner would have found it¹. As a result, the step length parameter, α_i , is initialised to a value in the range: $0 < \alpha_i < 1$. With this step size, the algorithm iterates until a new iterate, \mathbf{Q}_{i+1} , is generated that contains a collision. Having encountered such a collision, the algorithm computes a linearised constraint based on the point of collision, inserts the new constraint to the optimisation problem's constraint matrix, and then performs a new constrained iteration from the previous collision-free iterate, \mathbf{Q}_i . This new constrained iteration that is performed after updating the constraint matrix is performed, is performed with $\alpha_i = 1$. The advantage of doing this is that, with the constraints updated, if a collision-free is iterate generated using $\alpha_i = 1$ and the updated constraint matrix, then the minimum has been reached. Otherwise if a collision is detected when attempting to directly reach the minimum, then the algorithm reverts to moving smaller steps (i.e. $0 < \alpha_i < 1$) until a collision is detected, at which point the constraint matrix is updated and an attempt to directly reach the minimum under the new set of constraints is made. This process repeats, with the algorithm alternating between attempting to directly reach the minimum (each time constraints are updated) or moving in small steps, until the minimum is reached or until a path that is within a user-specified tolerance is found.

More compactly, the basic steps of the algorithm are as follows:

1. Compute a new iterate with the constraint matrix, Φ and α_i in the range: $0 < \alpha_i < 1$.
2. Verify if the new iterate is collision-free.
3. If the new iterate contains a collision, compute a collision constraint and add it to the constraint matrix, Φ .

¹The path planners developed in this thesis are modified so that they start by attempting a direct connection from the initial configuration to the goal configuration before they begin the actual planning process. This is done to avoid unnecessarily wasting time planning the path to the goal, when such a simple connection would have accomplished the task.

4. Backtrack to the previous collision-free solution.
5. Compute a new iterate with the newly updated constraint matrix, Φ and $\alpha_i = 1$.
6. Verify if the new iterate is collision-free.
7. If the new iterate contains a collision, go to step 1, otherwise the minimum has been reached.

Algorithm 19 gives pseudocode for the algorithm. It starts by initialising the gradient descent step length or learning rate (line 1) and a variable to be used to indicate if the minimum has been reached, `minReached` (line 2). It then iterates, gradually reducing the path length until a path with the minimum possible cost is found. In each iteration, the LCQP optimal step, $-H^{-1}\nabla L(\mathbf{Q}_i)^T$, is computed by the `computeOptimalStep()` function. The new solution is then computed and checked for collisions. If no collisions are detected, the current solution is updated to be the new, collision-free solution. Otherwise a collision constraint is added to the optimisation problem and the current solution is not updated – effectively retaining the previous collision-free solution, i.e. backtracking. In each subsequent iteration, the process repeats, either with a newly updated path or with an updated constraint matrix. The algorithm terminates immediately once a collision-free iterate is generated with $\alpha_i = 1$ under the updated set of constraints or if LCQP optimal step, is less than a given threshold (lines 3 and 5).

Algorithm 19: `gradientBasedPathOptimisation(node path Q)`

```

1  $\alpha_i \leftarrow \alpha_{\text{init}}$  ; // Initialise descent step length
2 minReached  $\leftarrow$  false ; // No solution yet
3 while not(noCollision and minReached) do
4    $\mathbf{P} \leftarrow \text{computeOptimalStep}()$  ; // LCQP optimal step
5   minReached  $\leftarrow$  ( $\|\mathbf{P}\| < 10^{-3}$  or  $\alpha_i == 1$ ) ; // Solution found
6    $\mathbf{Q}_{i+1} \leftarrow \mathbf{Q}_i + \alpha_i \mathbf{P}$  ; // New set of path nodes
7   if not(collisionFree( $\mathbf{Q}_{i+1}$ )) then
8     noCollision  $\leftarrow$  false ; // New solution contains collisions
9     if ( $\alpha \neq 1$ ) then
10      computeCollisionConstraint( $\mathbf{Q}_{i+1}, \mathbf{Q}$ ) ; /* Compute collision constraint based
11        on new path and previous collision-free path */
12      findNewConstraint() ; // Find constraint that is linearly independent
13      addCollisionConstraint() ; // Insert constraint to constraint Jacobian matrix
14       $\alpha_i \leftarrow 1$  ; // Revert to attempting to directly reach the minimum in one step
15    else
16       $\alpha_i \leftarrow \alpha_{\text{init}}$  ; // Revert to small steps if collisions have been detected
17  else
18     $\mathbf{Q} \leftarrow \mathbf{Q}_{i+1}$  ; // A new, collision-free path has been found
19    noCollision  $\leftarrow$  true;
19 return  $\mathbf{Q}$ 

```

An illustration of results for the application of the algorithm to path optimisation for a path planned for 2-dimensional robot in an obstacle-cluttered environment is shown in Figure 2.22. As shown in the figure, straight interpolations were used to connect node configurations in Campana et al.'s work – the input and output paths are both concatenations of straight lines and as such discontinuous in curvature. As a result, to be used in this project, the algorithm has to be adapted for continuous-curvature path optimisation.

The next subsection concludes our review of path-optimisation algorithms by selecting a suitable gradient-based path-optimisation technique and detailing how it will be adapted and used in the project.

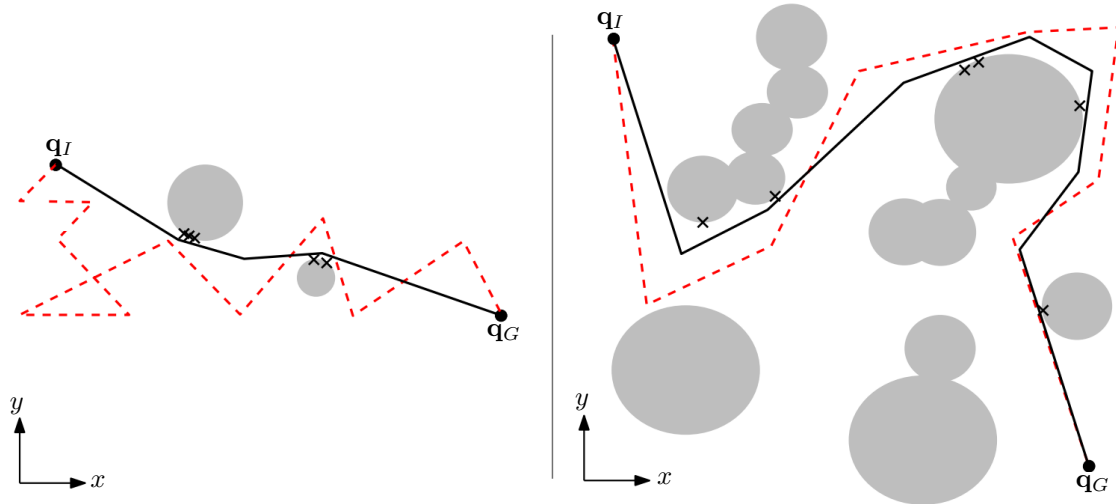


Figure 2.22: Path optimisation results for a two-dimensional robot navigating an environment with obstacles represented by gray obstacles in the work of Campana et al. [136]. The initial path is shown as the red, dotted line while the optimised path is given by the solid black line. The crosses represent obstacle contact points which have led to collision constraints. The scenario on the left shows an initial node path which has a self-intersection. The gradient-based path optimiser succeeds in removing this self intersection. The scenario on the right illustrates an application of the path optimiser in the presence of more obstacles.

2.8.2.1 Conclusion on Gradient-based Path Optimisation

This section has reviewed existing gradient-based path-optimisation algorithms. Among the reviewed algorithms, one was found that directly optimises path length. This is Campana et al.'s algorithm. In addition to directly optimising path length, unlike its contending counterparts, this algorithm also does not require any geometry preprocessing or any other offline computations to account for constraints, but it adds them to the path optimisation problem as they are encountered, online. While this algorithm undoubtedly stands out among its contending counterparts, in its current form, it optimises paths composed of straight lines and also produces paths that are a concatenation of straight lines. It is in this regard that the algorithm does not meet the path optimisation requirement of this project, which is continuous-curvature path optimisation. As a result, in this thesis, the algorithm will be adapted for continuous-curvature path optimisation and then be incorporated into the path planners selected in Subsection 2.6.6, to accelerate their rate of convergence. The versatility of the adapted algorithm in accelerating the convergence of the path planners towards the optimal solution will then be compared to that of the shortcut-based path-optimisation algorithms of Section 2.8.1.

At this point, we have reviewed both shortcut-based and gradient-based optimisation techniques for reducing the path of a given node path. The following subsection concludes our discussion on path optimisation with a summary of our findings.

2.8.3 Conclusion on Path-optimisation Techniques

Two categories of path-optimisation techniques have been considered, namely shortcut-based and gradient-based optimisation. Among shortcut-based path-optimisation techniques, considered algorithms include the path pruning method, the random shortcut method and the wrapping process. A decision has been taken to adapt all of these shortcut-based path-optimisation algorithms for continuous-curvature path optimisation and incorporate them into the path-planning algorithms selected in Section 2.6.6 to accelerate the rate of convergence of the planners. This allows us to compare the effectiveness of each path optimiser in accelerating the rate of convergence relatively to the effectiveness of its counterparts. Among gradient-based path-optimisation algorithms, a more recent algorithm has been selected for adaptation for continuous-curvature path optimisation and for use in the acceleration of the rate of convergence of the selected path planners. This is Campana et al.'s algorithm. Among reviewed gradient based path-optimisation techniques, this

algorithm stood out since it directly optimises path length – the same optimisation objective as this project. It also does not require any geometry preprocessing nor does it perform any offline computations. All these attributes of the algorithm are consistent with the requirements of this project.

The selected path-optimisation algorithms are adapted for continuous-curvature path optimisation in Section 5.5 and they are incorporated to the selected path planners for the acceleration of the rate of convergence in Section 5.6.

This concludes our review of literature for optimised path planning. The following chapter reviews techniques for path tracking, which is the process by which an autonomous vehicle autonomously executes a planned path produced by the techniques discussed in this chapter.

Chapter 3

Path Tracking

The preceding chapter discussed methods for planning a path that takes an autonomous vehicle from its initial configuration to a specified goal configuration. An important requirement was that the produced paths should satisfy the particular vehicle's motion constraints. The reason for that is that for the planning to be useful, the vehicle should be able to follow or execute the planned path – otherwise the planning efforts would be futile. A class of path-planning algorithms satisfying this requirement has been selected and extensions to it have been proposed. With a suitable path-planning algorithm in place, it becomes necessary to develop an algorithm for autonomously executing the planned paths. The review of algorithms that serve this purpose is the subject of this chapter. These algorithms are known as *path trackers*, *path-tracking controllers* or *path-tracking algorithms*.

3.1 Elements of a Path Tracking Algorithm

Path tracking refers to the process of executing a planned path by applying appropriate linear and angular velocity commands that guide the vehicle along the path [142]. The goal is to cause the vehicle to follow the planned path accurately so as to avoid exposing it to the risk of collisions. During planning, the planned path is ensured to be collision-free; deviation from this path during execution violates this guarantee. This section focuses on the elements of a path-tracking algorithm.

A path-tracking algorithm is essentially a control system and as such, its basic elements are simply the basic building blocks of a control system. These blocks are: the state of the system, the dynamics (or model) of the system, a reference input, the system output, a controller or control law and a control signal. The state of the system describes its present configuration and it can be used to determine the future behaviour of the system given excitation inputs and equations describing the system dynamics. The system dynamics or model describes how the state of the system changes over time. A controller is responsible for generating a control signal, which influences the behaviour of the system towards desired behaviour. The desired behaviour is given by the reference input. A control system may or may not use a measure of the system output in generating the control signal. If the system output is not used, then the system is known as *open-loop control*, otherwise it is called *closed-loop* or *feedback control*. Besides using the system output, a control system may make use of predictions in generating the control signal. A relevant example is that of driving a car [29, p. 2], where looking at the road ahead provides some valuable information for *predictive control*. Such information that anticipates the track to be followed or a possible disturbance can be used to *feed-forward*, an early warning to the control system, resulting in a predictive control strategy as opposed to a reactive one.

With the basic components of a controller introduced and controller classification briefly discussed, we now specifically consider the components of a path tracker and different classes of path trackers. In the context of a path tracker, the reference input corresponds to the path being tracked. This path is basically a curve in the configuration space that connects the initial configuration to the goal configuration as generated by the path-planning techniques of the preceding chapter. This path can be defined as continuous function; however, in most practical cases it is discretised either into a series of straight line segments or as a series of closely-spaced path nodes [143]. These different path representations are illustrated in Figure 3.1.

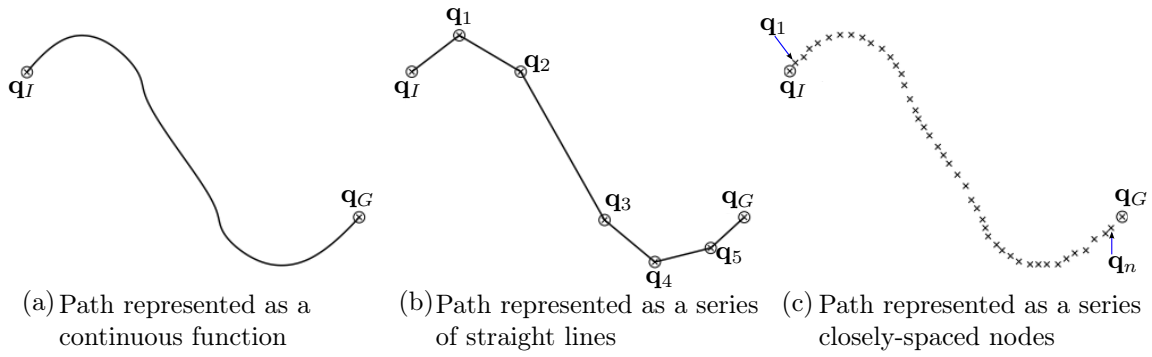


Figure 3.1: Various ways of describing a path. All these paths connect the same configuration pair (q_I, q_G) . The difference is in how they are represented.

In a path-tracking controller, the state of the system is the position and orientation (pose) of a reference point on the vehicle. The choice of this reference point is made by the system designer, with the centre of the vehicle's rear axle, the centre of the front axle, or the vehicle's centre of gravity being some commonly-used reference points. Estimates of state information may be obtained for example from onboard sensors such as GPS, IMU and odometers or from a SLAM system that uses cameras and/or LiDAR sensors. For purposes of path tracker control law design, a model of the vehicle is required. This may be a geometric model, a kinematic model or a dynamic model [21]. Control laws for path trackers based on a geometric model are formulated by exploiting the geometric relationships between the vehicle model and the path being tracked. Typically a simple geometric model of the vehicle known as the geometric bicycle model is used. The bicycle model simplifies the model of a four-wheeled vehicle by collapsing the four wheels into two wheels, each located at the centre of each axle – like a bicycle as shown in Figure 3.2. With the reference point being the position of the centre of the rear axle, this abstraction results in a simple geometric relationship, $\tan(\delta) = \frac{L}{R}$, between the steering angle of the front steerable wheel, δ , the wheelbase (i.e. the distance between the two wheels), L , and the radius, R , of the circular arc that the centre of the rear axle will follow. It is this simple geometric relationship that geometric path trackers exploit. Generally, any reference point can be chosen and the corresponding relationship between the model parameters and the planned path derived.

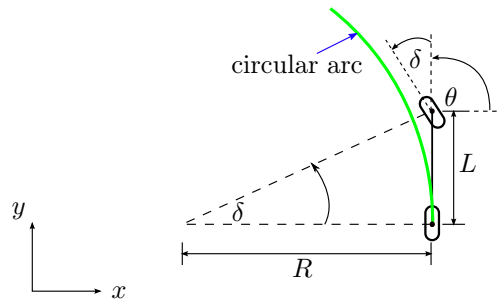


Figure 3.2: The geometric bicycle model. This model simplifies the model of a four-wheeled vehicle into a two-wheeled model, with one wheel at the centre of each axle. With the reference point being the position of the centre of the rear axle, the path followed by the model for a steering angle δ is circular arc of radius $R = \frac{L}{\arctan(\delta)}$.

Path trackers based on the kinematic model also make use of the bicycle model, but in this case, instead of a geometric relationship between the front and rear wheel, equations of motion are derived using wheel velocities. This is known as the kinematic bicycle model and it is shown in Figure 3.3. It enables simple vehicle motion analysis and the derivation of intuitive control laws based on wheel velocities. As it will become evident in Section 3.3, path tracking using the kinematic model can also be referred to as *path tracking with position-based feedback*, since

the position and the orientation of the wheels is used to compute position and orientation errors, which are then used in conjunction with the velocities to form the control law that controls the vehicle along the planned path.

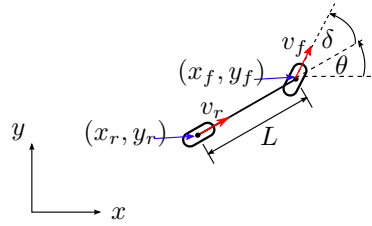


Figure 3.3: The kinematic bicycle model. This model simplifies the model of a four-wheeled vehicle into a two-wheeled model, with one wheel at the centre of each axle. Instead of a geometric relationship between model parameters, equations motions involving the wheel velocities are used to derive control laws.

Path trackers based on the dynamic model of the vehicle extend the bicycle model to not only include kinematics, but also dynamics of the vehicle – i.e. not only velocities, but also the forces that cause these velocities. We consider path trackers based on geometric and the kinematic model because they have low computational requirements [144] and are simple to understand and implement, yet have been demonstrated to be effective, having been successfully applied in a number of successful autonomous vehicle projects [21]. Section 3.2 presents a review of geometric path tracking techniques and Section 3.3 presents path trackers based on the kinematic model.

3.2 Geometric Path Trackers

Geometric path trackers are one of the earliest [144] and most popular [21] classes of path-tracking methods. They make use of the geometric relationship between the vehicle and path being tracked. They work by fitting a curve that starts from the vehicle's current configuration to a point on the reference path that is a distance, l_d , called the *lookahead distance* ahead of the vehicle. Intuitively, by making the vehicle follow this curve, the path tracker causes the vehicle to regain the path at the *lookahead point* in case of deviation. These path tracking methods are characterised by the fact that they do not explicitly attempt to follow the planned path, but instead, they chase the lookahead point. For this reason they are also known as *pursuit-based path-tracking methods* since they can be thought of as pursuing the ever-moving lookahead point. We now consider three classes of geometric path trackers, namely head-to-goal controllers, follow-the-carrot, as well as pure-pursuit and its variants.

3.2.1 Head-to-goal Path Tracker

The head-to-goal path tracker is the most basic pursuit-based path tracking method. It is applicable to a path that can be represented as a sequence of waypoints. It makes use of a PID head-to-goal controller that causes the vehicle to head directly towards the next upcoming waypoint by controlling the heading error, θ_e , between the vehicle's current heading and the heading to the upcoming waypoint. This process is repeated until the vehicle reaches the goal. In the context of the definition of geometric path trackers, the curve that this method fits is a straight line and the lookahead point is always fixed to exactly correspond to the upcoming waypoint. It is illustrated in Figure 3.4.

This path-tracking controller is the simplest among the reviewed path trackers. However, it is also the least accurate. It is incapable of accurately tracking the path even in the absence of disturbances. Once the vehicle deviates from the planned path, it does not attempt to track the path, but instead it heads directly towards the next waypoint, without regard to the planned path segment from the previous waypoint to the next waypoint. Even in the absence of disturbances, the path tracker cannot exactly track the planned path due to the discontinuous switch in desired

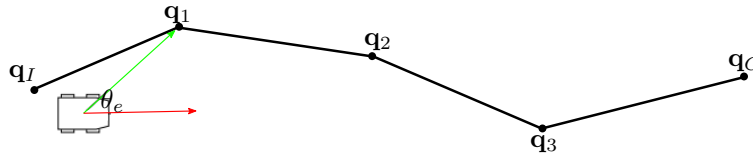


Figure 3.4: An illustration of the head-to-goal pursuit-based path tracker. The red arrow represents the vehicle's current heading and the green one is the heading from the vehicle's position to the upcoming waypoint. The heading error, θ_e , is the difference between the heading from the vehicle's position to the upcoming waypoint and the vehicle's current heading. The controller works by applying PID control on this error.

heading when switching from the current waypoint to the next waypoint – essentially attempting to reach the next waypoint by following a straight line from the current robot position. Moreover, since the method essentially fits a straight line from the vehicle's position to the target waypoint, it cannot accurately follow a path with a continuous-curvature profile. As a result, this path-tracking controller is unsuitable for the project. We now consider the follow-the-carrot path tracker which is an improvement to the head-to-goal path tracker.

3.2.2 Follow-the-carrot Path Tracker

The follow-the-carrot path tracker is a simple improvement on the head-to-goal path tracker. It improves path tracking achieved by the head-to-goal path tracker by making use of a fixed, user-specified lookahead distance as opposed to fixing the lookahead point to the next waypoint. As a result, the lookahead point (also known as the *carrot point*) pursued by the robot slides along the path at a distance equal to the lookahead distance in front of the robot. The principle behind follow-the-carrot originates from the idea of holding a carrot and moving away with it in front of a farm animal to coax the animal to follow you [145]. In geometric terms, follow-the-carrot also fits a straight line just like head-to-goal; however, the fitted line ends at the carrot point rather than the upcoming waypoint. In this case, the heading error, θ_e , is the difference between the heading to the carrot point and the robot's current heading. This is illustrated in Figure 3.5. A PID controller can again be used to reduce this error as in the head-to-goal path tracker. This path tracker exhibits improved tracking compared to its the head-to-goal path tracker as the robot attempts to adhere more closely to the planned path. This is because the carrot point can be at any point on the path segment between the previous waypoint and the upcoming waypoint, leading to the robot possibly regaining the path multiple times along this segment, depending on the length of the user-defined lookahead distance. This is opposed to just aiming to regain the path at the next waypoint like head-to-goal does.

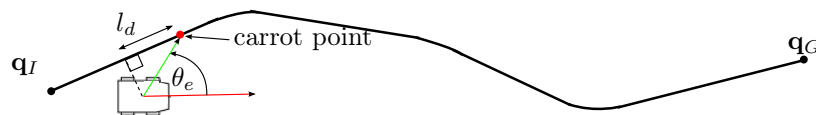


Figure 3.5: An illustration of the follow-the-carrot pursuit-based path tracker. The red arrow represents the vehicle's current heading and the green one is the heading from the vehicle's position to the carrot point. The lookahead distance, l_d , is user-specified and the heading error, θ_e , is the difference between the heading to the carrot point and the vehicle's current heading. The path tracker works by applying PID control on this error.

Just like the head-to-goal path tracker, the follow-the-carrot path tracker is easy to understand and implement. However, it has a number of drawbacks. Firstly, using this path tracker, a robot will cut corners, since it always follows a straight line from its current position to the carrot point. The tendency to cut corners worsens with a large lookahead distance. Secondly, the robot may oscillate about the planned path as it pursues carrot points. This particularly happens when a

small lookahead distance is used or when the robot moves at high speeds. Therefore, a great detail of attention has to be paid in choosing the lookahead distance as well as the robot's speed and in tuning the PID controllers to realise satisfactory tracking performance with minimal oscillations. Also, just like the head-to-goal path tracker, follow-the-carrot path trackers cannot accurately follow a path with a continuous-curvature profile, even in the absence of disturbances. This is it aims straight for the carrot point and ignores the intermediate path segment. As such, it is also unsuitable for use in this project. We now move on to the next, better pursuit-based path tracker, the pure-pursuit tracker.

3.2.3 Pure-pursuit Path Tracker

Pure pursuit is one of the most popular approaches to the path tracking problem [21], yet perhaps one of the earliest path-tracking algorithms – its origins can be traced back in history to missile target pursuit [146]. In the field of robotics, pure pursuit was originally applied by Wallace et al. [147] from the NavLab within Carnegie Mellon University's Robotics Institute. In their work, they used pure pursuit for estimating the steering angle necessary to maintain a robot on the road. The first account in which pure pursuit was used for a mobile robot to track an explicit path was in the masters thesis of Amidi [148] from the same institute in 1990. Then, after seeing the success of the algorithm, Coulter [149], again from the same institute, published a formal derivation and implementation of the algorithm two years later. Since then, pure pursuit became widely used for path tracking in mobile robotics. The pure-pursuit algorithm computes the curvature of a circular arc that the robot must follow in order to get back on the planned path at the lookahead point. It therefore works the same way as follow-the-carrot except that the curve fitted starting from the robot's current position and ending at the lookahead point is now a circular arc instead of a straight line. This computation is performed iteratively, with the lookahead point sliding along the path as the robot moves. An illustration of the path tracker is given in Figure 3.6. Each time the arc is computed, its curvature is used to determine the appropriate steering command to be given to the robot in order for it to track this arc. This process repeats iteratively with the lookahead point sliding along the planned path until the robot reaches the goal configuration.

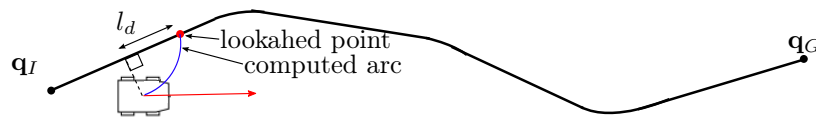


Figure 3.6: An illustration of the pure-pursuit path tracker. The red arrow represents the vehicle's current heading. The red dot is the lookahead point and the lookahead distance is l_d . The curve computed by the algorithm is a circular arc coloured blue. The curvature of this arc is used to deduce the steering command to be fed to the robot.

The pure-pursuit algorithm described above is superior to the head-to-goal and follow-the-carrot path trackers due to the use of the circular arc as opposed to a straight line. This is because, the arc computed by pure pursuit does not only take into account the geometry between the position of the vehicle and the planned path, but also the geometry of the steering of the robot. Pure pursuit is also easier to implement and tune than head-to-goal and follow-the-carrot, since it only has one parameter – the lookahead distance, as opposed to the latter algorithms, which have the PID controller gains in addition to the lookahead distance as parameters. Performance characteristics of the pure pursuit path tracker are adjusted by simply tuning the lookahead distance. With a smaller lookahead distance, the robot returns to the path more aggressively after deviation; however such aggression is often accompanied by oscillations. On the other hand, there are a number of benefits for having a longer lookahead distance. Firstly, it reduces oscillations. Secondly, a larger lookahead distance translates to a circular curve with a larger turning radius, which in turn translates to changes in commanded steering commands that are less abrupt. Thirdly, for paths that have sharp turns, a large lookahead distance allows the robot to begin turning before reaching the sharp turn, resulting in a smoother trajectory. However, this means the robot will cut such corners and not track them exactly. Lastly, a large lookahead distance results in less overshoot when the

robot returns to the planned path from a significant deviation. From these considerations, the pure pursuit algorithm provides significant improvements compared to head-to-goal and follow-the-carrot path trackers. The *vector pursuit*, which is discussed next, was introduced as a further improvement to path tracking performance achieved through pure pursuit.

3.2.4 Vector-pursuit Path Tracker

The vector-pursuit path-tracking algorithm was introduced in the PhD thesis of Wit [145, 150] in 2000. It improves the pure-pursuit algorithm by enforcing the condition that the orientation of the computed arc at the lookahead point must be the same as the orientation of the planned path at that point. To achieve this, their algorithm makes use of *screw theory* which was introduced by Sir Robert S. Ball in 1900 [151]. In motivating the use of screw theory in path tracking, Wit states that screw theory is a natural and appropriate tool for the representation of the desired motion of a rigid body, such as an autonomous vehicle, from its current position and orientation to a desired position and orientation. By enforcing that the orientation of the curve followed by the robot when attempting to regain the path matches the orientation of the planned path at the lookahead point, vector pursuit addresses an issue associated with pure pursuit, follow-the-carrot and head-to-goal path trackers wherein the robot immediately deviates from the path again after regaining it, if there is an orientation mismatch between the robot and the path. This makes vector pursuit more robust and accurate than its predecessors.

While vector pursuit introduces an evident improvement to pure pursuit, its computational process involving screw theory is much more complex than pure pursuit and more parameters need to be tuned to attain good performance [152]. A method that achieves the same objective as vector pursuit, i.e. ensuring that the orientation of the path followed by the robot has the same orientation as the planned path at the lookahead point, while remaining less complex is using a continuous-curvature path instead of a circular arc for the curve fitted for the robot's position to the lookahead point. Such a method is superior to vector pursuit in that it does not only enforce a match in orientation at the lookahead point, but also a match in curvature. This path-tracking technique is discussed next.

3.2.5 Continuous-curvature Path Tracker

The continuous-curvature pursuit-based path tracker was introduced by Girbés et al. [153, 154] who applied it in path tracking for unmanned ground vehicles, in particular industrial forklifts. The technique computes a continuous-curvature path from the current position and orientation of the robot to the lookahead point. The computed path is generated such that its orientation and curvature at the lookahead point is the same as the orientation and curvature of the planned path at that point. Such a continuous-curvature (CC) path generated for this purpose is shown in Figure 3.7. In Girbés et al.'s work, the cited benefit of using CC paths for path tracking is the comfort and safety experienced by people or goods being transported by the autonomous vehicle. An unmentioned benefit, which is important for this project, is that the robot's kinematic constraints, namely bounds on both the angular velocity and angular acceleration, are taken into account by this technique, while this is not the case with its predecessors, namely vector pursuit, pure pursuit, follow-the-carrot and head-to-goal.

The planned paths tracked by the ground robot in Girbés et al.'s work were a concatenation of straight lines (hence a discontinuous-curvature profile) as shown in Figure 3.7. With the CC path from the robot's current configuration to the lookahead point computed, the path tracking controller works by applying the curvature and sharpness (i.e. rate of change of curvature) profiles for a time duration, T , which is known as the kinematic control period. This process is repeated until the goal configuration is reached, with the lookahead point sliding along the planned path in front of the robot. Although in the original use of the algorithm, it was used on planned paths with a discontinuous-curvature profile, we believe that its usefulness can be greatly exploited in a setting where the planned paths are curvature continuous, such as this project. In such a setting, in the nominal case (i.e. when the robot is on the planned path), the curvature and sharpness profiles of the planned path can be applied and in the case of deviation, a CC path from the robot's

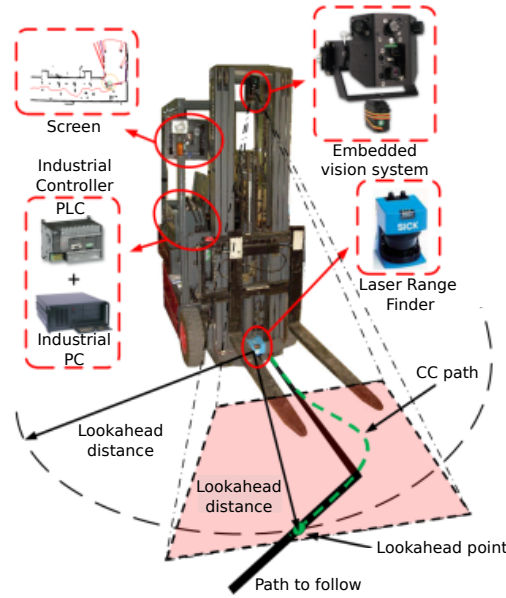


Figure 3.7: Continuous-curvature (CC) path tracking (Source: Girbés et al. [154]). The planned path is a concatenation of straight lines as shown. CC path tracking works by generating a CC path that starts at the robot's position and ends at the lookahead point and then applying motion commands that cause the robot to follow the CC path in the ideal case.

current position to the lookahead point can be computed and its curvature and sharpness profiles applied.

Among geometric path trackers, the continuous-curvature path tracker is most suitable for this project due to its above-mentioned attributes, namely enforcing the orientation and curvature at the lookahead point, satisfying kinematic constraints, and ensuring that in the absence of disturbances the planned path can be accurately tracked. It is important, however, to reiterate the qualification of its suitability, i.e. that it only results in accurate tracking in the absence of disturbances. In other words, the path tracker is an open-loop control strategy and can be used as a feed-forward controller. This concludes our treatment of geometric path trackers. In the next section we consider a different class of path tracker, namely those based on position feedback.

3.3 Path Tracking with Position-based Feedback

The path-tracking techniques reviewed in this section differ from those considered by the previous section in that they are based on the kinematic bicycle model rather than the geometric one. In particular, these techniques take a reference pose (position and orientation) on the robot as the controlled variable and use the measurement of that pose as an output to be used together with the planned path to derive a feedback control law that causes the reference point follow the planned path. With the pose of a reference wheel of choice (either that of the front wheel, (x_f, y_f, θ_f) , or that of the rear wheel, (x_r, y_r, θ_r)), path-tracking control is essentially achieved by using this together with the desired pose on the planned path to compute position and orientation errors and then adjusting the vehicle's steering by a function of these errors and the wheel velocity – the control law. In the following two subsections, we consider two techniques that are used for this purpose, namely *rear-wheel position-based feedback* and *front-wheel position-based feedback*.

3.3.1 Path Tracking with Rear-wheel Position-based Feedback

In path tracking with rear-wheel position-based feedback, the rear-wheel position is picked as the reference point on the kinematic bicycle model for feedback control law design. Kinematic

equations of the kinematic bicycle model with the reference point, (x, y) , chosen to be the position of the rear wheel (i.e. $x = x_r$ and $y = y_r$) are as follows [21, 31]:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\delta} \end{bmatrix} = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ \left(\frac{\tan(\delta)}{L}\right) \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \dot{\delta}, \quad (3.1)$$

where linear velocity is v_r is the linear velocity of the rear wheel. A convenient trick for path tracking is to express the kinematic bicycle model with respect to the planned path, which we desire to track. This is known as expressing the model in *path coordinates* [31]. The usefulness of this approach lies in that it enables us to properly define variables to be used in feedback control. Figure 3.8 shows the setup for path tracking with rear-wheel position-based feedback and the feedback variables used by the path-tracking technique. The path is parametrised by its arc length with the parameter, s , being the path parameter.

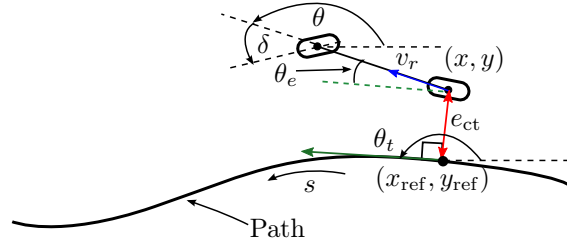


Figure 3.8: Path tracking with rear-wheel position-based feedback. The kinematic bicycle model of Figure 3.3 is used to track a planned path with the pose of the rear wheel as the controlled variable. A reference point, $(x_{\text{ref}}, y_{\text{ref}})$, on the path that is closest to (x, y) is used to compute feedback variables. These are the cross-track error, e_{ct} and the heading error, θ_e . The heading error is the difference between the orientation of the vehicle and orientation of the path tangent, θ_t and the cross-track error is the shortest distance from the position of the rear wheel to the path.

A reference point $(x_{\text{ref}}, y_{\text{ref}})$ on the path that is closest to the position of the rear wheel is selected and used to define the feedback variables. These feedback variables are the *cross-track error*, e_{ct} , and the *orientation error*, θ_e . The cross-track error essentially measures the lateral distance from the position of the rear wheel to the path reference point, $(x_{\text{ref}}, y_{\text{ref}})$. With the unit vector tangent to the path at $(x_{\text{ref}}, y_{\text{ref}})$ given by \mathbf{t} , and the vector from (x, y) to $(x_{\text{ref}}, y_{\text{ref}})$ given by \mathbf{d} , the cross-track error is simply the cross product between these two vectors:

$$e_{\text{ct}} = \mathbf{d} \times \mathbf{t}. \quad (3.2)$$

The orientation error is simply the difference between the robot's orientation and the angle of the path tangent at $(x_{\text{ref}}, y_{\text{ref}})$:

$$\theta_e = \theta - \theta_t(s). \quad (3.3)$$

The curvature, $\kappa(s)$, at length s along the path is given by:

$$\kappa(s) = \frac{d\theta_t}{ds}, \quad (3.4)$$

implying:

$$\dot{\theta}_t(s) = \kappa(s)\dot{s}. \quad (3.5)$$

Expressions for \dot{s} and \dot{e}_{ct} can be derived from Figure 3.8 as:

$$\dot{s} = v_r \cos(\theta_e) + \dot{\theta}_t e_{\text{ct}}, \quad (3.6)$$

$$\dot{e}_{ct} = v_r \sin(\theta_e). \quad (3.7)$$

By substituting the s , e_{ct} and θ_e for x , y and θ in Equation 3.1, kinematic equations for the rear-wheel drive kinematic bicycle model are as follows:

$$\begin{bmatrix} \dot{s} \\ \dot{e}_{ct} \\ \dot{\theta}_e \\ \dot{\delta} \end{bmatrix} = \begin{bmatrix} \frac{\cos(\theta_e)}{1 - \dot{e}_{ct}\kappa(s)} \\ \sin(\theta_e) \\ \left(\frac{\tan(\delta)}{L} - \frac{\kappa(s)\cos(\theta_e)}{1 - \dot{e}_{ct}\kappa(s)} \right) \\ 0 \end{bmatrix} v_r + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \dot{\delta}. \quad (3.8)$$

With this model in place, control techniques can be used to formulate a feedback control law that causes the robot to follow the planned path. Experimental results presented by Snider [21] show that the rear-wheel position-based feedback path tracker tracks planned paths accurately at low speeds. Particular mention is made that this path tracker outperforms pure pursuit – the most frequently used geometric path tracker. This versatility of rear-wheel position-based feedback can be attributed to the incredible ability of feedback control to handle process uncertainties, disturbances and modelling errors, which geometric path trackers lack. A drawback of the path tracker is that path tracking performance degrades with an increase in speed. In the same work, it is also noted that the path-tracking technique has trouble in tracking paths that have some discontinuities in curvature. However, since in our project planned paths are curvature continuous, this would not be an issue. When compared with front-wheel position-based feedback path tracking, which is the subject of the next subsection, Snider found that rear-wheel position-based feedback is not as good, especially at higher driving speeds.

3.3.2 Path Tracking with Front-wheel Position-based Feedback

In path tracking with front-wheel position-based feedback, the kinematic bicycle model is used to model the vehicle just as in rear-wheel position-based feedback. The difference is in that in this case, the reference point on the vehicle is chosen to be the pose of the steered front wheel. The approach was first used in Stanford University's robotic car, Stanley, which won the 2005 DARPA Grand Challenge [155]. The control strategy uses the variables s , e_{ct} and θ_e as in the preceding section. The difference is that in this case, the cross-track error, e_{ct} , is computed using the front wheel position instead of the rear wheel position. Figure 3.9 shows the setup for path tracking using front-wheel position-based feedback along with the variables used for feedback control.

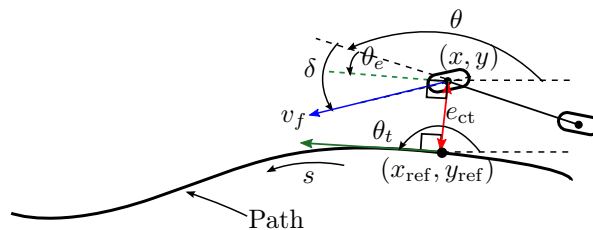


Figure 3.9: Path tracking with front-wheel position-based feedback. The kinematic bicycle model of Figure 3.3 is used to track a planned path with the position of the front wheel as the controlled variable. A reference point, (x_{ref}, y_{ref}) , on the path that is closest to (x, y) is used to compute feedback variables. These are the cross-track error, e_{ct} and the heading error, θ_e . The heading error is the difference between the orientation of the vehicle and orientation of the path tangent, θ_t and the cross-track error is the shortest distance from the position of the front wheel to the path.

The particular control law used in the Stanford robot, Stanley, which is popularly known as the *Stanley method*, makes use of a nonlinear feedback function of the cross-track error to command the robot's steering, for which exponential convergence to zero cross-track error can be shown [155]. The resulting steering control law is given by:

$$\delta = \theta_e + \arctan\left(\frac{ke_{ct}}{v_f}\right), \quad (3.9)$$

where k is a gain parameter and v_f is the linear velocity of the front wheel. This is an intuitive steering control law [21], which, in the case of zero cross-track error, causes the orientation of the front wheels of the robot to match the orientation of the path. This is enforced by the first term of the control law. The second term comes into play as soon as the cross-track error becomes non-zero. This term essentially adjusts the steering angle in nonlinear proportion to the cross-track error, e_{ct} – the larger the cross-track error, the stronger the steering response towards the planned path. Thus, as the cross-track error increases, the effect of the second term is to turn the front wheels to point towards the path, resulting in convergence to the path that is only limited by the vehicle's speed [155].

Results presented by Snider [21] show that the Stanley method works quite well for most driving situations as compared to both the pure pursuit and rear-wheel position-based feedback path tracking. It is also suitable for higher robot speeds than both these path trackers. A well-known issue with the Stanley method is that it has trouble with tracking paths with discontinuities in curvature [21, 144]. However, in our application, this is not an issue since the planned paths to be tracked are curvature continuous.

This marks the end of the review of path tracking techniques based on position feedback. The following section concludes our path tracking literature review by providing a consolidated summary of the reviewed path tracking techniques and the selection of those to be used in this project.

3.4 Conclusion on Reviewed Path Tracking Techniques

This chapter has considered existing techniques for path tracking. Particularly, two classes of path tracking algorithms have been considered and reviewed. These are geometric path trackers and position-based feedback path trackers. Five path trackers based on the geometry between the robot's position and the planned path were reviewed. These are the head-to-goal, follow-the-carrot, pure-pursuit, vector-pursuit and the continuous-curvature path trackers. On the other hand, two path trackers based on position feedback were reviewed. These are the rear-wheel position-based feedback and front-wheel position-based feedback path trackers. Within each class of path trackers, comparisons were made as each path tracker was introduced. This section aims to summarise the findings of this path-tracking literature survey and to single out techniques that are suitable for the path-tracking requirements of this project.

In general, the reviewed geometric path trackers work by causing the robot to follow a curve that starts from its current position and ends at a lookahead point on the planned path. Save for the vector-pursuit and the continuous-curvature path trackers, the rest of the path trackers immediately deviates from the path again after regaining it, due to orientation mismatch between that of the planned path and that of the vehicle at the point of regaining the path. The vector-pursuit and the continuous-curvature path tracker eliminate this issue by enforcing the condition that the orientation of the vehicle and that of the planned path should be the same at the point of regaining the path. The continuous-curvature path tracker is superior to vector pursuit in that it also ensures a curvature match at the point of regaining the path – an attribute that vector pursuit lacks. Vector pursuit is also more computationally involved. While in the reviewed literature the continuous-curvature path tracker was used to track planned paths with a discontinuous-curvature profile, causing it to fail to follow the planned path exactly, our project plans paths that are continuous in curvature, which can be accurately followed by the path tracker in the absence of disturbances.

Reviewed path trackers that are based on position feedback were found to be versatile in correcting for deviation from the planned path due to disturbances, with the front-wheel position-based feedback path tracker working well in most driving situations, including higher robot speeds than rear-wheel position-based feedback. It was noted that the front-wheel position-based feedback path tracker has issues tracking paths with a discontinuous-curvature and that this issue would not prevail in this project since planned paths are curvature continuous.

For the above reasons, the continuous-curvature path tracker has been selected as a feed-forward or open-loop control strategy in this project to let the robot track the path accurately in the nominal case, with a technique based on position-based feedback that is inspired by front-wheel position-based feedback used for correcting for deviations in the presence of disturbances.

This marks the end of our review of path tracking techniques and Part I of the thesis. At this point, existing techniques for both optimised path planning and path tracking have been reviewed, evaluated and appropriate ones selected for each sub-problem. Part II focuses on the design and implementation of optimised path-planning and path-tracking algorithms based on the choices made through the literature review.

Part II

Development of Algorithms, Analysis and Results

Part II details the design and implementation of the algorithms for solving the optimised path-planning and path-tracking sub-problems of the autonomous navigation problem. The first two chapters of this part focus on the formulation of the optimised path-planning algorithm. Chapter 4 solves the local path-planning problem, while Chapter 5 uses the results of the former as well as other techniques such as incremental search techniques, collision detection and optimisation to solve the optimised global path-planning problem. Finally, the design and implementation of an algorithm to solve the path-tracking problem is the subject of Chapter 6.

Chapter 4

Development of the Local Path Planner

This chapter details the development of the local path-planning algorithm. This is an algorithm used to find a feasible path between two configurations for the given vehicle model (in this case vehicles with a constrained angular velocity and angular acceleration) in the *absence* of obstacles. A feasible path is one that satisfies the vehicle's motion constraints. The problem of finding feasible paths in the absence of obstacles is known as the local path-planning problem. It is local in the sense that it only considers local constraints, i.e. constraints on the vehicle's motion, without considering global constraints (those outside the vehicle model), such as obstacles. The handling of global constraints is the task of a higher-level planner known as the global path planner, which has access to the local path planner and a collision detector. In this project, the global path planners to be developed are sampling-based, specifically, the anytime RRT and the informed RRT*, as concluded in Chapter 2 of the literature review. Both these global path planners will make use of the local path planner developed in this chapter. The following section explains the context of the local path planner within these global path planners. The development of the global path planners is the subject of Chapter 5.

4.1 The Context of the Local Path Planner

An illustration of the context of the local path planner within an RRT-based global path planner is given in Figure 4.1. As shown in this illustration, an RRT-based global path planner works by growing a tree of reachable configurations rooted at the initial configuration, \mathbf{q}_I , to search the configuration space for a solution path. At each iteration, the tree is grown by using random sampling of the configuration space to generate a new configuration to be added on the tree. An attempt to add the generated configuration to the tree is then made through a call to the local path planner, which computes a local path from an existing node on the tree to the new configuration. If the computed local path is collision-free, then the new configuration is added as a node on the tree and the collision-free local path is added as the branch connecting the newly added node to the tree. Otherwise, if the computed local path collides with obstacles, the new configuration is discarded and the algorithm proceeds to the next iteration. The tree is grown until it reaches the goal configuration (or gets arbitrarily close to it), or until the allocated number of iterations runs out. If the tree reaches the goal configuration or gets arbitrarily close to it, a solution path from \mathbf{q}_I to \mathbf{q}_G can be found by searching the tree. Otherwise, if the allocated iterations elapse without the tree reaching the goal configuration, failure is reported.

Through this brief and high-level consideration of the principle of operation of the global path planner, it is clear that the local path planner is called within the global path planner whenever it is required to determine a feasible path between two configurations – the new configuration and the existing node on the tree from which a connection of the new configuration is to be attempted. Therefore, algorithmically, the local path planner takes two arguments – the configuration pair between which the local path is to be computed. It has one return value – a structure variable representing the computed local path that is to be returned to the calling function, the global path planner.

With the context of how the local path planner fits into the bigger path-planning scheme discussed, we now proceed to the development of the local path-planning algorithm.

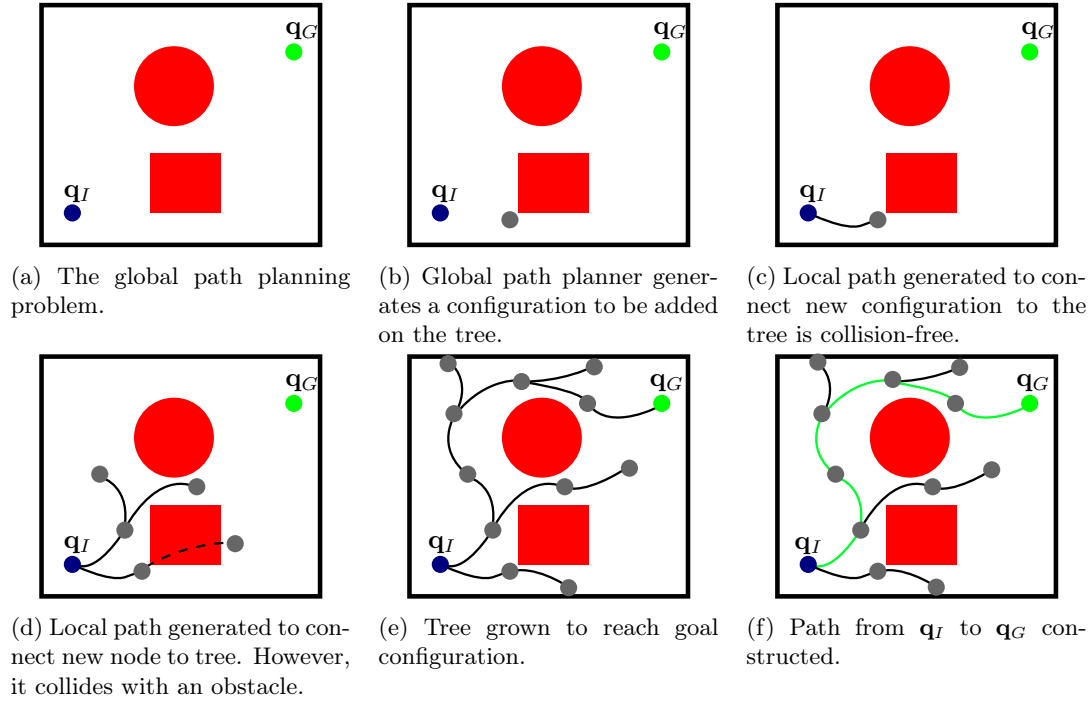


Figure 4.1: The context of the local path planner within the RRT-based global path planner. The global path planner builds a tree of reachable configurations, rooted at q_I . At each iteration, it generates a new configuration to be added to the tree (e.g. sub-figure (b)) and then makes a call to the local path planner, which generates a local path that forms a branch from an existing node on the tree to the new node. If this local path is collision-free, the new configuration is added to the tree (e.g. sub-figure (c)), otherwise it is discarded (e.g. sub-figure (d)). The tree is grown until it reaches the goal configuration (sub-figure (e)), at which point a solution path that is collision-free is found (coloured green in sub-figure (f)).

4.2 Local Path-planning Algorithm Development

This section considers the development of the local path-planning algorithm. As mentioned in the preceding section, the algorithm takes two configurations from the configuration space and computes a local path connecting them. In robotics literature, the local path-planning algorithm is usually referred to as the *local path-planning method* and abbreviated LPM. In this thesis we use these terms interchangeably. The LPM can be viewed as a function with function prototype: `localPath LPM(q_{start}, q_{end})`, where the arguments q_{start} and q_{end} respectively represent the configurations from which the local path is to start and end. The call of a local path-planning algorithm is known as a *local path-planning query*.

While it would be sufficient to simply return one local path for a given local path-planning query, the local path-planning algorithm developed in this thesis returns a set of feasible local paths connecting the configuration pair, (q_{start}, q_{end}) , sorted according path length to facilitate the picking of the best collision-free local path by the global path planner for the given configuration pair. The benefit of returning such a set of feasible local paths, as opposed to only returning the shortest, is that it gives the global path planner other options to choose from, in case the shortest path contains collisions. What makes this possible is that the local path planner does not perform collision checks for the paths it computes; instead, the collision checks are handled by the global path planner through a call to the collision detector.

An important realisation that makes the task of local path planning intuitive and manageable is that, in essence, a local path can be represented by the sequence of manoeuvres that the vehicle would be commanded to execute to move from q_{start} to q_{end} . We refer to a local path planner based on this principle as a *manoeuvre-based local path planner* and the principle behind it is introduced

in the following subsection.

4.2.1 The Concept of a Manoeuvre-based Local Path Planner

The task of the local path planner is to compute a set of feasible paths connecting a configuration pair $(\mathbf{q}_{\text{start}}, \mathbf{q}_{\text{end}}) \subset \mathcal{C}$, if such a set of paths exists, without considering possible collisions. Otherwise, if no such path exists, the target configuration, \mathbf{q}_{end} , is deemed unreachable from $\mathbf{q}_{\text{start}}$, and the local path planner reports failure. This action of the local path planner can be viewed as applying a change in the state or configuration, $\mathbf{q} \in \mathcal{C}$, of the vehicle by giving a control input $u \in \mathcal{U}$ according to the following general equation of motion:

$$\dot{\mathbf{q}} = f(\mathbf{q}, u), \quad (4.1)$$

with \mathcal{C} and \mathcal{U} being the configuration space and the control space respectively. For nonholonomic systems such as those addressed by this thesis, the dimensions of \mathcal{C} are more than those of \mathcal{U} , which means that the system has fewer control inputs than states. As such, these systems cannot follow arbitrary paths in the configuration space. The best way to control them is to define a manoeuvre space, \mathcal{M} – the set of all viable manoeuvres. A manoeuvre, picked from the set of all viable manoeuvres, represents the motion caused by applying a predefined control input, $u \in \mathcal{U}$, over a period of time, which leads to a change in the vehicle's state that adheres to the motion constraints of that vehicle. For the vehicles targeted by this thesis, manoeuvres refer to *straight*, *transition-from-straight-to-turn*, *turn*, or *transition-from-turn-to-straight* motion commands. This set of manoeuvres is informed by the different motions that a vehicle with a constrained angular velocity and angular acceleration is capable of performing to move from one configuration to another. In general, due to the bound on angular velocity, this motion would be a combination of straight-line movements and turns. However, due to the bound in angular acceleration, the switch from straight-line motion to turning motion (and vice versa) is not instantaneous, hence the need for the transition manoeuvres.

Each path (also known as a manoeuvre sequence) is then composed of a continuous combination of path segments or manoeuvres. The benefits of this manoeuvre-based local path planning include the following [22, 156]:

1. The nonholonomic or dynamic constraints of the vehicle are encoded into each manoeuvre, thus removing the need to check if each computed local path is dynamically feasible.
2. The use of manoeuvres reduces the number of variables needed to describe a motion command, dramatically reducing the complexity of designing input motion commands.

Borrowing the famous machine learning adage of *no free lunch*, we indeed do not get the above benefits for free. The cost that comes with the use of manoeuvres in local path planning is that it is difficult to determine a manoeuvre set that can fully describe the dynamic capabilities of the vehicle, while at the same time maintaining a low-dimensional manoeuvre space. We thus introduce an action space, which is a subset of the manoeuvre space, to reduce the complexity of the system. As an example, while a vehicle can execute all turns with a radius, r , within the range $r_{\min} \leq r \leq \infty$, with r_{\min} being its minimum turning radius¹, to reduce the complexity of the system, it is usually preferred to only allow turns of radius r_{\min} . Therefore, while it is entirely possible to plan local paths using a general (or variable) turning radius, the norm of using a specific chosen radius has remained dominant in the path planning community since the days of Dubins [107], who was the first to characterise local paths for vehicles with a constrained angular velocity. The reason is that it reduces the complexity of the local path planner. In consistency with this practice, the action space used in this project includes all the manoeuvres previously stated (i.e. *straight*, *transition-from-straight-to-turn*, *turn* or *transition-from-turn-to-straight* motion commands), with the exception that turns are limited to those of minimum turning radius instead of all possible turns. The next subsection discusses implications imposed on the local planner by the manner in which the configuration pair $(\mathbf{q}_{\text{start}}, \mathbf{q}_{\text{end}})$ is represented by the calling function – the global path planner – in a local path-planning query.

¹It is general practice to choose a value of r_{\min} that is slightly greater than the actual minimum turning radius of the vehicle during planning so as to allow some headroom for the control system that will track the planned path.

4.2.2 Local Path-planning Query Specification: Implications for the Local Path Planner

Prior to proceeding to the actual development of a local path-planning algorithm, one more thing that is necessary to understand is how the configuration pair $(\mathbf{q}_{\text{start}}, \mathbf{q}_{\text{end}})$ will be represented in a local path-planning query. In the context of sampling-based global path planning, which is the setting in which our local path planner will be used, the vehicle's initial configuration and goal configuration $(\mathbf{q}_I$ and $\mathbf{q}_G)$ are each given as a “position and orientation” vector (i.e. $\mathbf{q}_I = [x_I \ y_I \ \theta_I]^T$ and $\mathbf{q}_G = [x_G \ y_G \ \theta_G]^T$) to the sampling-based global path planner. Each path to the goal that the sampling-based global path planner finds is a series of intermediary configurations or waypoints, $\{\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_K\}$ between \mathbf{q}_I and \mathbf{q}_G . For reasons that will become apparent in the discussion of the design and implementation of the sampling-based global planner, it is preferable to specify each of these intermediate configurations as a “position only” vector, $[x_k \ y_k]^T$ (where $k \in \mathbb{Z} : k \in [1, K]$ is the index of the intermediary configuration in the list), with the orientation at each configuration treated as a “don't care” condition. The path curvature at both the initial and goal configuration as well as at each intermediary configuration, \mathbf{q}_k , is constrained to be zero in consistency with existing local path-planning techniques covered in Section 2.7. Thus the continuous-curvature path from the start to the goal must be generated from configurations or waypoints represented as a set, \mathbf{Q} , of the form:

$$\mathbf{Q} = \left\{ [x_I \ y_I \ \theta_I]^T, [x_1 \ y_1 \ \theta_{1(DC)}]^T, [x_2 \ y_2 \ \theta_{2(DC)}]^T, \dots, [x_K \ y_K \ \theta_{K(DC)}]^T, [x_G \ y_G \ \theta_G]^T \right\}, \quad (4.2)$$

where each element of the set is a configuration in the form $[x_k \ y_k \ \theta_k]^T$, specifying the position and orientation of each waypoint. Every $\theta_{k(DC)}$ entry signifies a “don't care” orientation condition at the particular configuration.

The task of the local path planner is to compute the path between each consecutive pair of configurations in \mathbf{Q} , starting from the initial configuration. Considering the nature of the configurations in \mathbf{Q} , two cases exist, namely: a case where $\mathbf{q}_{\text{start}}$ is specified as a “position and orientation” vector while \mathbf{q}_{end} is specified as a “position only” vector and one in which both $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} are specified in position and orientation. This might not seem obvious at first, due to the fact that all the intermediary configurations between \mathbf{q}_I and \mathbf{q}_G have “don't care” orientations. As a result, at first, it might seem as if there are two additional cases for the configuration pair: one where both $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} are specified in “position only” (the case where both $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} are both intermediary configurations), and another where $\mathbf{q}_{\text{start}}$ is specified in “position only” and \mathbf{q}_{end} is specified in “position and orientation” (the case where $\mathbf{q}_{\text{start}}$ is an intermediary configuration and $\mathbf{q}_{\text{end}} = \mathbf{q}_G$). These two cases, however, do not exist, because when the global path planner connects the waypoints with local paths, it starts by connecting the initial configuration to the first intermediary configuration, through the LPM call: $\text{LPM}([x_I \ y_I \ \theta_I]^T, [x_1 \ y_1]^T)$; the LPM call outputs the orientation by which the target configuration is reached (in this case θ_1). Thus the $\theta_{k(DC)}$ s are determined on each LPM call and incrementally get updated for each target intermediary configuration until the last intermediary configuration is reached. After this, the last call to the LPM is made, which connects the last intermediary configuration to the goal configuration. This is through the function call: $\text{LPM}([x_K \ y_K \ \theta_K]^T, [x_G \ y_G \ \theta_G]^T)$, since $\theta_{K(DC)}$ would have been updated to a known value, θ_K , after the call of $\text{LPM}([x_{K-1} \ y_{K-1} \ \theta_{K-1}]^T, [x_K \ y_K]^T)$ to determine the path by which \mathbf{q}_K is reached.

From this, we see the need to formulate two versions of the local path planner. One version requires $\mathbf{q}_{\text{start}}$ to be specified as a “position and orientation” vector, while \mathbf{q}_{end} is specified as a “position only” vector. This version is henceforth called the *continuous-curvature local path planner with “don't care” goal orientation* and shortened *CC LPM with “don't care” goal orientation*. The second version requires both $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} to be specified as “position and orientation” vectors. We refer to it as the *continuous-curvature local path planner with standard inputs* and shortened *CC LPM with standard inputs* in subsequent discussions. The development of these two versions of the local path planner is the subject next subsection.

4.2.3 Development of the Local Path Planners

In the preceding subsections, we introduced the concept of a manoeuvre-based local path planner, subsequently looking at the advantages of using such an approach for local path planning. We also considered the format in which a configuration pair query will be passed to the local path planner from the global path planner during a typical local path-planning query. This consideration helped us to understand the need for the two versions of the local path planner, namely the *continuous-curvature local path planner with “don’t care” goal orientation*, shortened, *CC LPM with “don’t care” goal orientation* and the *continuous-curvature local path planner with standard inputs*, shortened, *CC LPM with standard inputs*. This subsection is dedicated to the formulation of these local path planner versions as manoeuvre sequences. We begin by discussing the construction of primitive manoeuvres or motion primitives, and then proceed to discuss the formulation of each version of the LPM as a concatenation of these primitives.

4.2.3.1 Motion Primitives for Continuous-curvature Paths

As stated in Section 4.2.1, in the context of vehicles with a constrained angular velocity and angular acceleration, a manoeuvre can be one of the following motion primitives:

1. straight
2. transition-from-straight-to-turn
3. turn
4. transition-from-turn-to-straight

A “straight” motion primitive is a case where the orientation of the vehicle does not change during the course of the motion, i.e. $\dot{\theta} = 0$, and the curvature of the path, κ , is zero. On the other hand, during a “transition-from-straight-to-turn” motion primitive, the curvature of the vehicle changes according to some function, f , from an initial curvature, κ_0 , to a final curvature, κ_{\max} . The nature of this function depends on the model or dynamics of the actuator used to influence the curvature of the vehicle over time². An example of such an actuator is a steering wheel in a car-like vehicle. As stated in Section 2.7.3, there are a number of curves to consider for this purpose, but a clothoid, which results in curvature being a linear function of path length, is an attractive option. During a turning manoeuvre, the curvature is held at a constant non-zero value, i.e. $\kappa = \dot{\theta} = \text{constant}$. This characterises a circle of radius $r = \frac{v}{\dot{\theta}}$, with v being the linear velocity of the vehicle. Lastly the “transition-from-turn-to-straight” manoeuvre is symmetric to the “transition-from-straight-to-turn”.

In summary, these manoeuvres can be geometrically represented by:

1. a straight line
2. a clothoid with an initial curvature $\kappa_0 = 0$ and maximum curvature $\kappa_{\max} = \frac{1}{r}$
3. a circular arc of radius r
4. a clothoid with an initial curvature $\kappa_0 = \kappa_{\max}$ and minimum curvature $\kappa = 0$

Thus geometrically, the local path planning task is to construct a set of paths connecting $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} , where each path is a concatenation of a continuous combination of clothoid, circular arc and straight line segments and it is curvature-continuous everywhere. Finally, the local path planner must sort this set according to path length.

To facilitate a better understanding and visualisation of the construction of continuous-curvature paths through the concatenation of the various motion primitives described above, we start by giving an example of such a path, generated by our algorithm. This example is shown in Figure 4.2(a)

²This change in curvature over time is equivalent to a change in curvature with path length for a vehicle moving at a constant velocity

and its discontinuous-curvature counterpart is as shown in Figure 4.2(b). The purpose of this illustration is to give a high-level overview of how continuous-curvature paths look like and how they differ from those with a discontinuous-curvature profile. The actual development of the two identified versions of the continuous-curvature local path planner follows in Sections 4.2.3.2 and 4.2.3.3, after this example and its accompanying discussion. This discontinuous-curvature path is a Dubins path, it has been developed by our developed Dubins path planner. As stated in Subsection 2.7.4, Dubins local path planner has been selected for the local path planning aspect of this project and in this chapter, it is adapted for curvature continuity. The continuous-curvature path has been generated by the adapted Dubins local path planner.

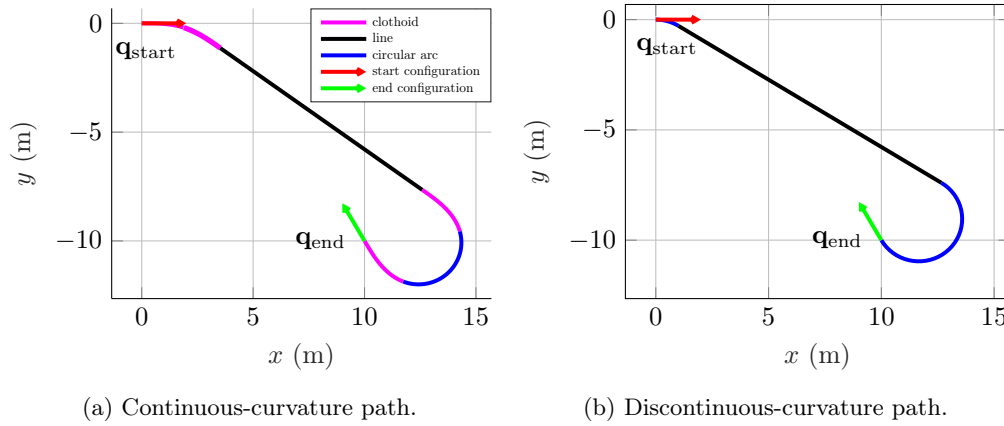


Figure 4.2: An example of local path planning for a local path-planning query $\text{LPM}([0 \ 0 \ 0]^T, [10 \ -10 \ \frac{2\pi}{3}]^T)$ through the concatenation of motion primitives introduced above. In (a), a continuous-curvature local path is shown, while (b) shows the discontinuous-curvature counterpart.

In this example, the discontinuous-curvature path begins with a circular arc (with curvature, $\kappa = \frac{1}{r_{\min}}$) followed by a straight line ($\kappa = 0$) and ends with an arc with curvature, $\kappa = \frac{1}{r_{\min}}$. It is important to note a number of areas of curvature discontinuity on this path. First, the initial configuration of the vehicle is tangent to the first circular arc of curvature $\kappa = \frac{1}{r_{\min}}$ and it has zero curvature. This exhibits a curvature mismatch between the vehicle and the path at its beginning. The second point of discontinuity is between the initial turn and the straight-line manoeuvre. Again, the straight-line manoeuvre is tangent to the turning circle, resulting in an abrupt change from a non-zero curvature of $\kappa = \frac{1}{r_{\min}}$ to a zero curvature. This characteristic repeats at the exit from the straight line manoeuvre into the last turn. Then finally, at the goal configuration, there is a curvature mismatch similar to that which exists at the initial configuration between the curvature component of the configuration ($\kappa = 0$) and the path's curvature ($\kappa = \frac{1}{r_{\min}}$).

The continuous-curvature path, on the other hand, begins with a right-turning clothoid starting at an initial curvature, $\kappa_0 = 0$, to a final curvature, $\kappa < \kappa_{\max}$, followed by a symmetric left-turning clothoid of initial curvature, κ_0 , equal to the final curvature of the first clothoid, and the final curvature of the second clothoid is $\kappa = 0$. This clothoid is then followed by a straight line ($\kappa = 0$) and the line is followed by a right-turning clothoid starting with $\kappa_0 = 0$ and ending with curvature $\kappa = -\kappa_{\max}$, which is in turn followed by a circular arc of curvature $\kappa = -\kappa_{\max}$. Finally, the path is concluded by a left-turning clothoid with $\kappa_0 = \kappa_{\max}$ and $\kappa = 0$. It is our hope that this continuous-curvature example demonstrates two points, namely:

- the continuity in curvature everywhere along the path, and
- the fact that for some turns (soft turns), the maximum curvature, κ_{\max} is not reached, hence a circular arc motion primitive is not required, while for other turns (sharp turns), κ_{\max} is reached and a circular arc portion is required. Further details on when each of these two cases arise will be covered in the following subsections.

In the following subsection we discuss the development of the version of the continuous-curvature local path planner for which both $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} are given as “position and orientation” vectors.

4.2.3.2 Continuous-curvature Local Path Planner with Standard Inputs

As discussed in Subsection 4.2.2, for the problem at hand, two versions of the local path planner are required. One has a prototype or header of the form $\text{LPM}([x_{\text{start}} \ y_{\text{start}} \ \theta_{\text{start}}]^T, [x_{\text{end}} \ y_{\text{end}}]^T)$ and we chose to call it the *continuous-curvature local path planner with “don’t care” goal orientation*, shortened *CC LPM with “don’t care” goal orientation*, while the other has a prototype of the form $\text{LPM}([x_{\text{start}} \ y_{\text{start}} \ \theta_{\text{start}}]^T, [x_{\text{end}} \ y_{\text{end}} \ \theta_{\text{end}}]^T)$ and we elected to refer to it as the *continuous-curvature local path planner with standard inputs*, shortened *CC LPM with standard inputs*. This name of the second version of the planner is the same as the title of this section and its development is discussed here, with the development of the first version dedicated to Subsection 4.2.3.3.

As discussed in the literature review, the problem of formulating a local path planner of the form $\text{LPM}([x_{\text{start}} \ y_{\text{start}} \ \theta_{\text{start}}]^T, [x_{\text{end}} \ y_{\text{end}} \ \theta_{\text{end}}]^T)$ was originally solved by Dubins [107] and followed up by Reeds and Shepp [108], both resulting in solutions with discontinuous-curvature profiles. The idea of continuous-curvature local paths simply builds up on these two seminal approaches to eliminate the curvature discontinuity problem. As a result, one way to develop a continuous-curvature local path planner is for one to first go through the development of the original local path planners and then adapt them to the continuous-curvature case. This is the approach used in this thesis. Again, in the conclusion of our local path planning literature review (Subsection 2.7.4), Dubins local path planner was selected as the local path-planning method to be adapted in this project. We therefore start off by describing our development of the Dubins LPM, and then proceed to adapt it, resulting in its continuous-curvature version.

4.2.3.2.1 Development of Dubins LPM with Standard Inputs

This subsection details our development of the standard Dubins LPM by following the procedure outlined by Dubins. The computation of the set of Dubins paths connecting $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} begins by computing two pairs of turning circles of radius r_{min} ; one pair at $\mathbf{q}_{\text{start}}$ and the other at \mathbf{q}_{end} . Each pair is constructed such that both circles have their radii tangent to the corresponding configuration’s velocity vector. Once these turning circles are computed, the next step is to connect them with tangent curves which are either straight lines or circles. Whether the connecting tangential curves are straight lines or circles depends on the Euclidean distance, D , between $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} . Specifically, if $D < 4r_{\text{min}}$, then the tangential components are circles of radius r_{min} , and the resulting path is a concatenation of three circular turn manoeuvres, denoted *CCC*. Otherwise, in the case of $D \geq 4r_{\text{min}}$, the tangential components are straight line tangents, with the resulting path being a concatenation of a circular turn, a straight line and a final circular turn manoeuvre and denoted *CSC*. More specifically, *CCC* paths can be either *LRL* or *RLR* and *CSC* paths can be one of *RSR*, *RSL*, *LSL* and *LSR*, where *L* denotes a left turn, *R* denotes a right turn and *S*, a straight line.

A more richer notation specifies each manoeuvre with its corresponding length and duration. Tables 4.1 and 4.2 show this notation for *CCC* and *CSC* paths respectively. The notation used is as follows for *CCC* paths: the subscript angles α , β and γ are used to characterise the respective angular distances of first, second and third turn respectively. A slightly similar notation is used to describe *CSC* paths, which is the following: the subscript angle α is used to represent the angular distance of the first turn, then the subscript d represents the length of the straight line manoeuvre and finally, the angle γ describes the angular distance of the last turn. The beauty of this notation is that it makes it explicitly clear how to go about computing each manoeuvre. Particularly, for circular arc manoeuvres, we compute the angles (α, β, γ) required for the turn, and from that the distance covered during the manoeuvre can be determined. In the straight line manoeuvre case, the straight line distance, d , is simply the length of the tangent line segment.

We now describe the process of computing the paths in the *CCC* and *CSC* cases more concretely, with examples. The task of computing viable *CCC* paths reduces to computing the respective tangential circles between a pair of initial-goal turning circles and using those circles to

Path type	Manoeuvre 1	Manoeuvre 2	Manoeuvre 3
RLR	R_α	L_β	R_γ
LRL	R_α	R_β	L_γ

Table 4.1: Dubins *CCC* paths and their manoeuvre specifications. This is the set of paths that exists if the Euclidean distance, D , between $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} is less than 4 times the minimum turning radius (i.e. $D < 4r_{\text{min}}$). Each manoeuvre is a circular arc, with R representing a right-turning arc and L denoting a left-turning arc. The angular distances of each arc are represented by the respective angles, α , β and γ . From these angles and the value of the minimum turning radius, computing the distance of each manoeuvre is straight-forward.

Path type	Manoeuvre 1	Manoeuvre 2	Manoeuvre 3
RSR	R_α	S_d	R_γ
RSL	R_α	S_d	L_γ
LSL	L_α	S_d	L_γ
LSR	L_α	S_d	R_γ

Table 4.2: Dubins *CSC* paths and their manoeuvre specifications. This is the set of paths that exists if the Euclidean distance D , between $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} is $D \geq 4r_{\text{min}}$. Each manoeuvre is either a circular arc or a straight line. Precisely, R denotes a right-turning arc, while S denotes a straight line and L represents a left-turning arc. The angles, α and γ are used to describe the angular distances of the first and last turns respectively. The length of the straight-line segment is characterised by the distance, d . Distances for the circular arcs are computed as previously mentioned, using the angles and the value of r_{min} .

determine the lengths of the respective manoeuvres or path segments of each path. This construction is shown in the example in Figure 4.3. Here, the left and right initial configuration turning circles have centres Ω_{il} and Ω_{ir} respectively while the left and right goal configuration turning circles have centres Ω_{gl} and Ω_{gr} respectively. The tangent circle between the left turning circles at the initial and goal configurations has a centre Ω_{tr} . This circle, together with the two left-turning circles, is used in the computation of the *LRL* path. The points of tangency in this case are marked \mathbf{q}_{1r} and \mathbf{q}_{2r} . They are used in conjunction with the centres of the three circles to compute the values of the angles α , β and γ , which are in turn used to compute the length of each arc segment and hence the overall length of the *LRL* path. Likewise, the tangent circle between the right turning circles at the initial and goal configuration has a centre Ω_{tl} . This circle, together with the two right-turning, is used in the computation of the *RLR* path. The points of tangency are now marked \mathbf{q}_{1l} and \mathbf{q}_{2l} and are likewise used in conjunction with the centres of the three circles the angles α , β and γ in the computation of the *RLR* path, and subsequently its length.

The task of computing viable *CSC* paths is similar to that of computing *CCC* paths illustrated above, except that in this case the tangential components that we compute are straight lines. Thus the computation of initial and goal configuration turning circles remains the same as shown in Figure 4.3, with the new computation being that of tangent lines connecting each pair of initial-goal turning circles instead of tangential circles. The left and right initial configuration turning circles have centres Ω_{il} and Ω_{ir} respectively, while the left and right goal configuration turning circles have centres Ω_{gl} and Ω_{gr} respectively. Tangent lines are computed between all combinations of initial-goal turning circles as shown in Figure 4.4. Since there are 4 circles, the total number of tangent lines is 16; however, only 4 of these lines result in the *CSC* manoeuvre sequences listed in Table 4.2. These are marked in solid lines, while the other, unusable tangents are dotted.

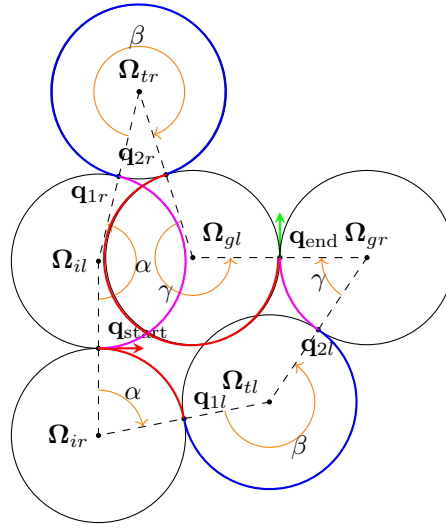


Figure 4.3: The geometry for the construction of Dubins *CCC* Paths. In this figure, the centres of the turning circles at the start configuration are Ω_{ir} and Ω_{il} respectively for the right and left turns. Similarly, centres of turning circles at the target configuration are Ω_{gr} and Ω_{gl} for the right and left turning circles. The circles centred at Ω_{tr} and Ω_{tl} represent the tangential circles used to compute the *LRL* and *RLR* paths respectively. Finally, the angles α , β and γ represent the angular distances for each turn as explained before. The resulting paths are marked as follows: the *LRL* path has its first turn coloured magenta, the second coloured blue and the last one red; on the other hand, the first turn of the *RLR* path is coloured red, the second coloured blue, and the last magenta. The reason for using different colours for the first and last turns of the two paths is to make them easily differentiable as they both start and end at the same point.

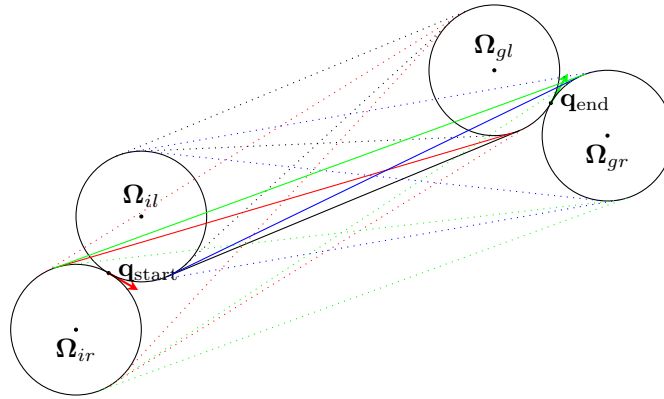


Figure 4.4: The geometry for the construction of Dubins *CSC* paths. The turning circles at the initial configuration have centres, Ω_{il} and Ω_{ir} respectively. The goal configuration has similar turning circles denoted Ω_{gl} and Ω_{gr} . Tangent lines are then used to connect each initial-goal turning circle pair. For each pair, there are 16 possible tangents, but only 1 results in a feasible path for that pair. The *RSR* tangent is coloured green, the *RSL* one, red, then the ones for the *LSL* and *LSR* paths are respectively marked black and blue.

The construction is taken a step further in Figure 4.5, where the turning circles and tangent line for each *CSC* case in Table 4.2 is independently presented. In each case, the point of tangency at the initial turning circle is denoted by \mathbf{q}_1 , while the similar point at the goal turning circle is denoted by \mathbf{q}_2 . These points are again used in conjunction with the centres of the associated turning circles to

determine the angles α and γ , and subsequently the length of each turning manoeuvre. The length of the straight-line manoeuvre is simply the Euclidean length of the tangent line. Figures 4.5(a), 4.5(b), 4.5(c) and 4.5(d) show this process for the *LSL*, *RSL*, *LSR* and *RSR* paths respectively.

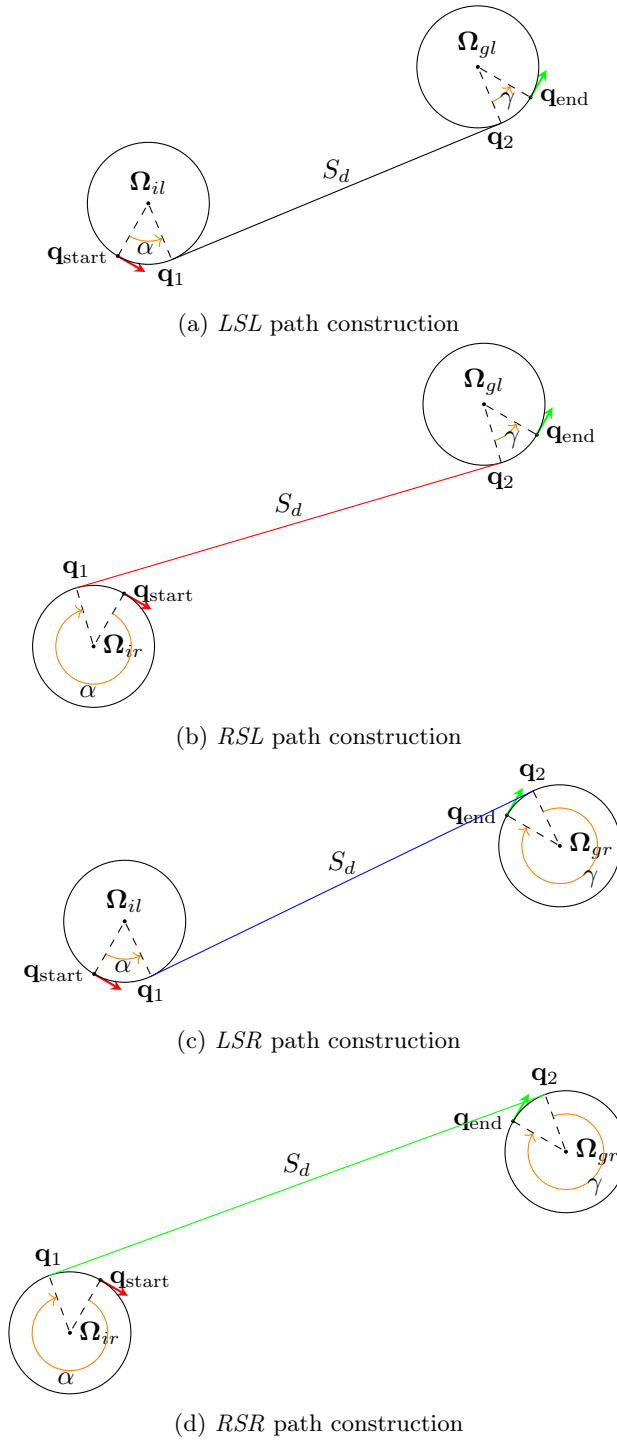


Figure 4.5: Isolation and construction of each *CSC* case. The colouring of tangent lines is consistent with that of Figure 4.4. This figure enhances Figure 4.4 by showing the angles of each turn, α , β and γ , as well as the tangent points.

At this point, the process for computing Dubins paths has been fully described geometrically.

By translating this geometric description to a computer algorithm, an implementation of the developed LPM is achieved. In this project, Dubins LPM with standard inputs has been written in MATLAB and C++. The following subsection presents results obtained by running the developed LPM for given local path-planning queries.

4.2.3.2.2 Results from Runs of Developed Dubins LPM with Standard Inputs

The developed Dubins LPM with standard inputs has been written in MATLAB and C++. In this project, MATLAB has been used for fast prototyping and testing of each developed algorithm. On the other hand, the translation to C++ has been done to make it possible to deploy the developed algorithms in the Robot Operating System (ROS) during integration with other constituting modules of the overall autonomous navigation system, which is the subject of Part III.

We show sample outputs of our MATLAB implementation in Figures 4.6 and 4.7 for local path-planning queries that result in *CCC* and *CSC* paths respectively. Figure 4.6 shows the solution local paths resulting from a run of our MATLAB implementation of the algorithm for the *CCC* case, sorted according to path length. This output also clearly shows the respective path segments in different colours as well as the inflection or manoeuvre-switching points, \mathbf{q}_1 and \mathbf{q}_2 .

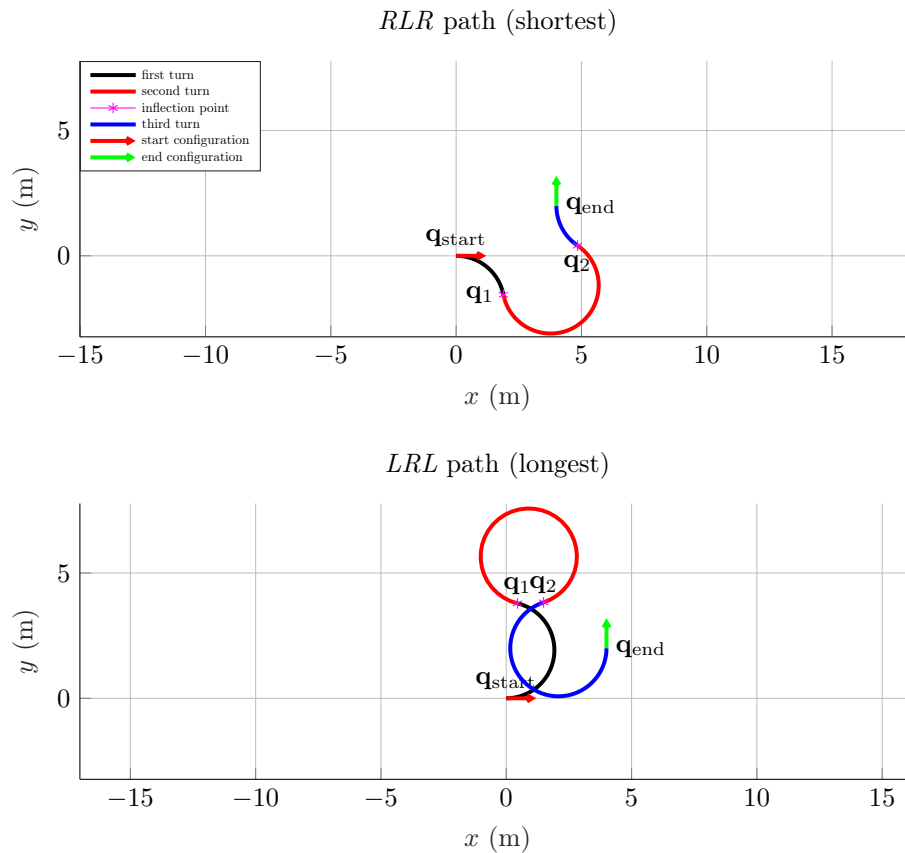


Figure 4.6: Computed *CCC* paths for a local path-planning query with $\mathbf{q}_{\text{start}} = [0 \ 0 \ 0]^T$ and $\mathbf{q}_{\text{end}} = [4 \ 2 \ \frac{\pi}{2}]^T$. The paths are sorted according to path length as shown, with the *RLR* path being the shortest and the *LRL* being the longest. The path segments or manoeuvres are marked as follows for both paths: the first turning manoeuvre is coloured black, the second, red and the third, blue.

For the *CSC* case, the resulting paths as generated by our MATLAB implementation of the developed algorithm are shown in Figure 4.7. The paths are again sorted according to path length, and the different path segments as well as manoeuvre-switching points are clearly marked.

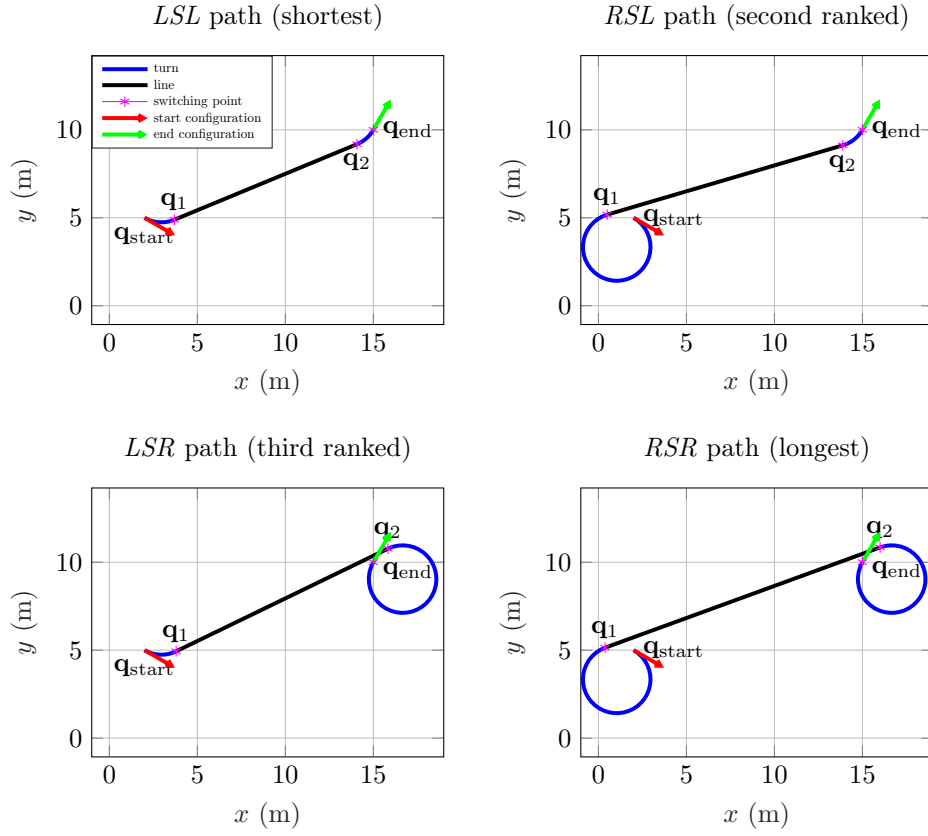


Figure 4.7: Computed *CSC* paths for a local path-planning query with $\mathbf{q}_{\text{start}} = [2 \ 5 \ -\frac{\pi}{6}]^T$ and $\mathbf{q}_{\text{end}} = [15 \ 10 \ \frac{\pi}{3}]^T$. The paths are sorted according to path length as shown.

We have now developed and demonstrated a working algorithm for the computation of a solution to a local path-planning query of the form $\text{LPM}([x_{\text{start}} \ y_{\text{start}} \ \theta_{\text{start}}]^T, [x_{\text{end}} \ y_{\text{end}} \ \theta_{\text{end}}]^T)$. This algorithm was developed based on the method introduced by Dubins [107] and although it solves the local path planning problem, the resulting paths are discontinuous in curvature. As stated in the introduction of this section, our approach is to start by developing this local path-planning method with a discontinuous-curvature profile, and then adapt it to the continuous-curvature case for the generation of paths that respect the curvature and sharpness constraints associated with vehicles with constrained angular velocity and angular acceleration. In the next section, we go through the adaptation part of this process for the version of the local path planner with standard inputs (i.e. both $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} specified as “position and orientation” vectors).

4.2.3.2.3 Adaptation of Dubins LPM with Standard Inputs to the CC LPM with Standard Inputs

The Dubins LPM with standard inputs, which was developed in the preceding subsection, suffers from a discontinuous-curvature profile at the points of inflection at transitions between different manoeuvres. For a vehicle with constrained kinematics to accurately follow such a path, one of the following conditions is required:

- The vehicle must stop at every point of discontinuity and re-orient its steering wheel. This is an undesirable behaviour in most practical, time-critical autonomous navigation missions.
- If the vehicle is to follow the path without stopping, it must be able to change its angular velocity instantaneously (i.e. it must have an infinite angular acceleration). This is impractical for vehicles targeted by this thesis as they have a bound on angular acceleration. The

practical implication then is that without stopping, such a vehicle will not be able to follow the planned path exactly, but it will deviate from it at the points of discontinuities.

Owing to the above-mentioned reasons, Dubins LPM is not sufficient in most practical autonomous navigation applications. It should be noted, however, that this LPM is in no way useless. Firstly, it can be directly used if enough headroom in terms of clearance from obstacles is allowed to account for the errors that would be incurred during path execution. In this manner, the advantage of the Dubins LPM, i.e. its simplicity, can be exploited, easing the computational cost of the planning process. Secondly, this implementation can be extended or adapted to a continuous-curvature version to account for the constraint in angular velocity and acceleration at the transition points. The latter approach is used in this thesis, as concluded in Subsection 2.7.4, and the development of the adapted LPM is presented here.

To ensure curvature continuity, Dubins paths are adapted by replacing the turning circles at the initial and goal configuration with special manoeuvre sequences called *CC turns*, short for *continuous-curvature turns*. These turns are an invention of Fraichard and Scheuer [110]. We source techniques from their work for adapting Dubins paths for continuous-curvature local path planning. In general, a CC turn essentially replaces a circular arc turning manoeuvre with a continuous combination of the following manoeuvres in the specified order:

- (a) A clothoid arc of sharpness $\sigma = \pm\sigma_{\max}$, whose curvature starts from $\kappa = 0$ and increases linearly to $\kappa = \kappa_{\max}$, with σ being the proportionality constant in this linear curvature increase. Here, σ_{\max} is the maximum allowable rate of change in curvature for the vehicle.
- (b) A circular arc of curvature $\kappa_{\max} = \frac{1}{r_{\min}}$, where r_{\min} is the vehicle's minimum turning radius.
- (c) A clothoid symmetric to the one in (a), of sharpness $-\sigma$, whose curvature starts from $\kappa = \kappa_{\max}$ and decreases linearly to $\kappa = 0$.

An example computation of a general CC turn³ is shown in Figure 4.8. This turn was generated by our MATLAB implementation of Fraichard and Scheuer's CC turn formulation. It will be used to explain the concept and properties of CC turns.

For the computation of the CC turns, we leverage the ability to compute the individual manoeuvres at the origin with zero orientation and then concatenate them by translating and rotating them in the plane as required. For this concatenation, it is very important for the manoeuvres to be fully specified in position *and* orientation, especially at the start and end points. As seen in the example, the computation of the turn begins with the computation of the first clothoid that starts at \mathbf{q}_1 , has initial curvature, $\kappa = 0$, sharpness, $\sigma = \sigma_{\max}$, and final curvature, κ_{\max} . Of great importance is the full specification of the configuration at the end of this clothoid, \mathbf{q}_a . This configuration is a function of the clothoid's sharpness as well as its maximum curvature and it is given by [110]:

$$\mathbf{q}_a = \begin{cases} x_a = \sqrt{\pi/\sigma_{\max}} C_f \left(\sqrt{\kappa_{\max}^2 / (\pi\sigma_{\max})} \right) \\ y_a = \sqrt{\pi/\sigma_{\max}} S_f \left(\sqrt{\kappa_{\max}^2 / (\pi\sigma_{\max})} \right) \\ \theta_a = \kappa_{\max}^2 / (2\sigma_{\max}) \\ \kappa_a = \kappa_{\max} \end{cases}, \quad (4.3)$$

with C_f and S_f being the Fresnel cosine and sine integrals. Since the curvature at \mathbf{q}_a is κ_{\max} , it lies on a circle of radius $r = r_{\min} = \kappa_{\max}^{-1}$, which is the same as the Dubins turning radius. The centre of this circle is given by:

$$\mathbf{\Omega} = \begin{cases} x_{\Omega} = x_a - r_{\min} \sin \theta_a \\ y_{\Omega} = y_a + r_{\min} \cos \theta_a \end{cases}. \quad (4.4)$$

The next path segment in the CC turn manoeuvre sequence is a circular arc segment of radius r_{\min} beginning at \mathbf{q}_a and centred at $\mathbf{\Omega}$, whose end point, $\mathbf{q}_b = [x_b \ y_b \ \theta_b \ \kappa_{\max}]^T$, is the same as the

³This is a left-turning CC turn; a right-turning one can be constructed similarly, by having a negative sharpness for the first clothoid as well as $\kappa_{\max} = -\frac{1}{r_{\min}}$.

An Example Continuous-curvature Turn

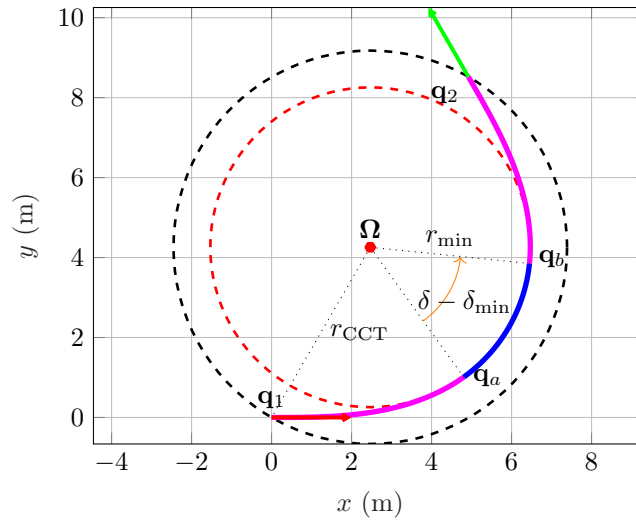


Figure 4.8: An example CC turn with $\mathbf{q}_1 = [0 \ 0 \ 0 \ 0]^T$, $\theta_1 = 0$ and $\theta_2 = \frac{2\pi}{3}$. The inner circle is of radius r_{\min} and it is exactly the same as the Dubins turning circle. The outer circle represents a locus of all possible starting points for a clothoid of sharpness, σ_{\max} , and initial curvature, $\kappa = 0$, if it has to terminate exactly on a point that is on the circumference of the Dubins turning circle. In the same manner, this outer circle is also the locus of all possible end points for an outgoing clothoid of sharpness, σ_{\max} , and initial curvature, κ_{\max} , and symmetric to the first clothoid.

starting point of the second clothoid and is dictated by the position of the end point of the turn⁴, \mathbf{q}_2 . For this reason, the second clothoid is computed first; after its placement, the configuration, \mathbf{q}_b , is determined and the circular arc is computed. This completes the computation of the CC turn, which is simply a concatenation of these three manoeuvres.

An important realisation is the existence of an outer circle called the *CC circle*, which is the locus of all possible end points of the second clothoid. This circle has a radius, $r_{\text{CCT}} = \sqrt{x_{\Omega}^2 + y_{\Omega}^2}$. Since the two clothoids are symmetrical, the start configuration, \mathbf{q}_1 , which is the start point of the first clothoid, is also located on the circumference of this circle. The centre of the CC circle is the same as that of the Dubins turning circle, Ω , given in Equation 4.4. The end configuration of the turn makes an angle $\mu = \text{atan}(x_{\Omega}/y_{\Omega})$ with the tangent of the CC circle and the start configuration makes an opposite of this angle, $-\mu$. This phenomenon is known as μ -tangency and will be used later for the placement of special tangential lines or circles connecting initial and goal CC circles in the computation of CC paths, using CC turns in a manner that is slightly different from the classical Dubins case.

The existence and length of each of the above-mentioned manoeuvres in a CC turn depends on how much change in orientation (also known as *deflection*, denoted δ) the turn is seeking to achieve. If we denote the configuration at the start of the CC turn as $\mathbf{q}_1 = [x_1 \ y_1 \ \theta_1 \ 0]^T$ and the configuration at the end of the turn to be $\mathbf{q}_2 = [x_2 \ y_2 \ \theta_2 \ 0]^T$, then the deflection, δ , is given by:

$$\delta = (\theta_2 - \theta_1) \bmod 2\pi. \quad (4.5)$$

All CC turns of all deflections can contain all the manoeuvres contained in the example CC turn, with each having a non-zero length; however, for certain specific cases, this leads to redundant or unnecessary turning manoeuvres, resulting in what is known as *self-intersecting CC turns* in which the CC turn makes a loop and intersects itself when δ is too small or too large.

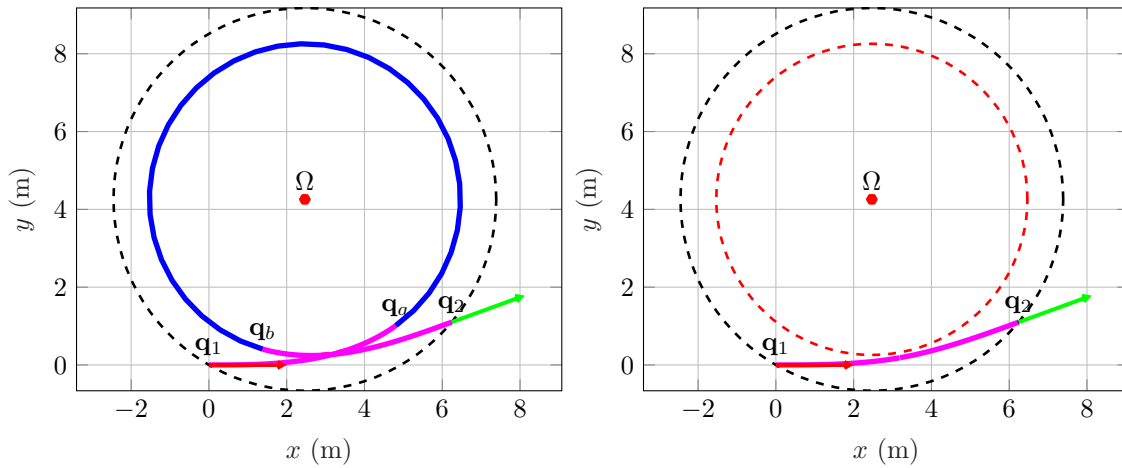
It is important to note two boundary or edge cases that can be excluded from self-intersecting turns and handled separately. These are: (a) when $\delta = 0$ and (b) when $\delta = \delta_{\min}$. The manner in which they are handled is as follows:

⁴The end point of the turn is in turn determined by the change in orientation that the turn is seeking to achieve.

- (a) $\delta = 0$: In this case, the start and end configuration of the turn (\mathbf{q}_1 and \mathbf{q}_2) have exactly the same orientation. As such, the turn is not necessary and it is simply replaced by a straight line from \mathbf{q}_1 to \mathbf{q}_2 .
- (b) $\delta = \delta_{\min}$: In this case, the circular arc in Figure 4.8 vanishes, and the turn is made up of the two symmetric clothoids of sharpness σ_{\max} and $-\sigma_{\max}$ respectively.

Self-intersecting turns thus only happen under two cases, namely: (a) when $0 < \delta < \delta_{\min}$ (i.e. values δ in-between the boundary cases discussed above) – turns falling in this category are said to be *turns of small deflections* and (b) when $\delta_{\min} + \pi \leq \delta < 2\pi$ – turns falling in this category are said to be *turns of large deflections*. These two cases are handled as follows:

- (a) $0 < \delta < \delta_{\min}$: In this case, self-intersection phenomenon is avoided by replacing the CC turn with two symmetric clothoids with a maximum curvature less than κ_{\max} , thus completely avoiding any circular arc manoeuvre. This is shown in Figure 4.9.



(a) An example CC turn with $0 < \delta < \delta_{\min}$. (b) The shorter equivalent CC turn for the CC turn in (a).

Figure 4.9: An example computation of a CC turn with δ in the range $0 < \delta < \delta_{\min}$. In this particular case, $\delta = \frac{\pi}{9}$. The general CC turn for this scenario with all manoeuvre components non-zero is shown in sub-figure (a) is clearly self-intersecting. The optimal equivalent path for the turn is shown in sub-figure (b), where the general CC turn is replaced by a pair of symmetric clothoids that never reach the minimum turning radius and thus completely avoid the unnecessary circular arc turn.

- (b) $\delta_{\min} + \pi \leq \delta < 2\pi$: One way of handling this self-intersection case is to replace it with a path that begins by forward-going clothoid of sharpness σ_{\max} up to configuration \mathbf{q}_a . Once \mathbf{q}_a is reached, the second manoeuvre is a backward circular turn of radius r_{\min} until \mathbf{q}_b is reached. The last manoeuvre is a clothoid of sharpness $-\sigma_{\max}$, which begins at \mathbf{q}_b with curvature, $\kappa = \kappa_{\max}$, and ends at the end configuration of the turn, \mathbf{q}_2 , with curvature, $\kappa = 0$. This is illustrated by Figure 4.10. In this thesis, we handle this self-intersection case without requiring reverse motion by realising that when a left-turning CC turn exhibits this type of self-intersection for the given deflection, then a right-turning CC turn would exhibit a self-intersection of the form in (a) for the same deflection (i.e. a small deflection). The same applies for the case where a right-turning CC turn has a large deflection – a left turning CC turn will have a small deflection. Through this realisation, when the left turning exhibits this form of self-intersection, the right turn with a smaller deflection can be used instead, thus saving the day – providing a shorter turn, with no reverse motion for the required deflection. The same happens when a right turn exhibits a large deflection, a left turn results in a

smaller deflection and saves the day. This will be seen in action in an illustration of CC path construction that follows a bit later in this subsection (Figures 4.12 and 4.13).

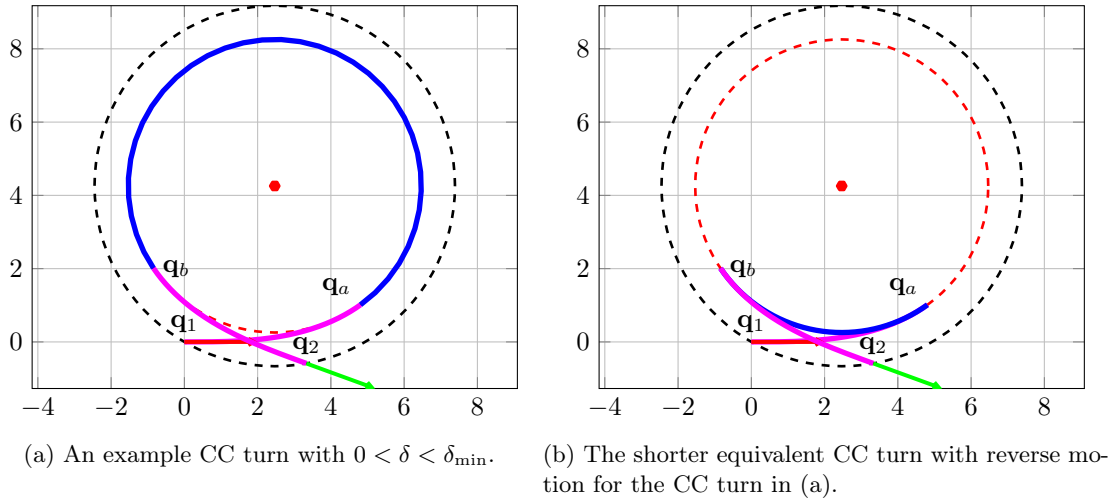


Figure 4.10: An example computation of a CC turn with $\delta - \delta_{\min} > \pi$. The general CC turn for this scenario with all manoeuvre components non-zero is shown in sub-figure (a) is clearly self-intersecting. The shorter equivalent turn for the self-intersecting turn is shown in sub-figure (b), where the original left-turning CC turn is replaced by right-turning CC turn, which is shorter than its left-turning counterpart.

Now that we have developed the algorithm to compute CC turns of any deflection, the next step is to make use of these turns to construct continuous-curvature local paths between a given configuration pair. The procedure for computing CC paths in this manner is similar to that used to compute Dubins paths; it starts with the computation a pair of circles at both the initial and goal configuration, but the difference is that instead of circles of radius r_{\min} , CC circles of radius r_{CCT} are constructed. Once these circles have been constructed, the next step is to establish the existence of tangential curves between each initial-goal CC circle pair. Similarly to the Dubins case, these curves are either straight lines or circles. The difference is that for CC circles the tangency relationship is a special one, known as μ -tangency. In this tangency relationship, a line is said to be μ -tangent to a CC circle if it crosses the CC circle so as to make an angle μ with the tangent at the intersection point. Just as in the classical tangency case, there are four possible μ -tangents between two CC circles, two being internal and the other two being external tangents when the distance between their centres is greater than $2r_{\text{CCT}}$. In the case where the separation is less than $2r_{\text{CCT}}$, the tangential curve is a third CC circle and the path is purely made up of turns.

With the procedure for constructing CC paths through the placement of tangential curves between the initial-goal CC circle pairs outlined, we proceed to illustrations of the construction process. We begin by illustrating the construction of CC paths, which are purely made up of turns – similar to Dubins *CCC* paths. This process is shown in Figure 4.11. The notation used for the turning circles follows the one used under the Dubins case with the right and left initial configuration turning circles having centres Ω_{ir} and Ω_{il} respectively, while the respective goal configurations are denoted by Ω_{gr} and Ω_{gl} . Tangential circles are then computed and denoted by Ω_{tr} and Ω_{tl} for the *LRL* and *RLR* paths. From these circles, the start and end configurations of all CC turns making up each path are determined as shown.

With the start and end of configuration of each turn determined, the computation of the continuous-curvature *LRL* and *RLR* paths reduces to using the start and end configuration of each CC turn to compute the deflection, δ , required for that turn and then using the deflection to determine the manoeuvres making up that CC turn. The CC turn is then a concatenation of the respective constituting manoeuvres. The *LRL* path is obtained by concatenating three CC turns, namely:

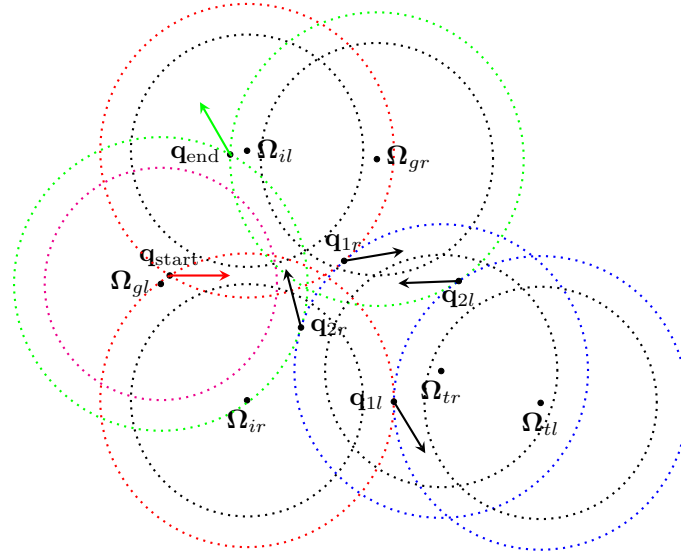


Figure 4.11: The geometry for the construction of continuous-curvature *RLR* and *LRL* paths. In this illustration, $\mathbf{q}_{\text{start}} = [0 \ 0 \ 0]^T$ and $\mathbf{q}_{\text{end}} = [1 \ 2 \ \frac{2\pi}{3}]^T$; they are respectively marked with red and green arrows indicating their position and orientation. The CC circles at the start configuration, $\mathbf{q}_{\text{start}}$, are coloured red while those at the end configuration, \mathbf{q}_{end} , are green. Tangent turning circles are coloured blue. Notation used for turning circle centres follows the one used in the Dubins case: Ω_{il} and Ω_{ir} are the centres for the left and right start turning circles, Ω_{gl} and Ω_{gr} are the left and right turning centres at the end configuration, and lastly, Ω_{tl} and Ω_{tr} are the respective centres for the tangent circles of the *RLR* and *LRL* paths. Each tangent turning circle has the configurations for the start and end of the turn (corresponding to \mathbf{q}_1 and \mathbf{q}_2 in the general CC turn of Figure 4.8) marked as follows: \mathbf{q}_{1l} and \mathbf{q}_{2l} for the *RLR* tangent circle, and \mathbf{q}_{1r} and \mathbf{q}_{2r} for the *LRL* tangent circle. For the start turning circles, $\mathbf{q}_{\text{start}}$ corresponds to the start configuration of the turn and the start configuration of the corresponding tangent circle marks the end configuration of the CC turn. In the case of the CC circles at the end configuration, \mathbf{q}_{end} , the end configuration of the associated tangent circle is the start of the turn and the end configuration, \mathbf{q}_{end} , marks the end of the turn. Using these configurations, the deflection for each CC circle can be determined and a corresponding CC turn computed. The concatenation of respective CC turns in the order from start to tangent to end results in *LRL* and *RLR* paths.

-
- (a) The left-turning CC turn at the start configuration, $\mathbf{q}_{\text{start}}$ (i.e. the one on the CC circle with centre Ω_{il}).
 - (b) The right-turning tangential CC turn on the CC circle, centred at Ω_{tr} , which is μ -tangential to both the left CC circle at $\mathbf{q}_{\text{start}}$ and the left CC circle at \mathbf{q}_{end} .
 - (c) The left-turning CC turn at the end configuration, \mathbf{q}_{end} (i.e the one on the CC circle centred at Ω_{gl}).

In a similar way, the *RLR* path is obtained by concatenating three CC turns, namely:

- (a) The right-turning CC turn at the start configuration, $\mathbf{q}_{\text{start}}$ (i.e. the one on the CC circle with centre Ω_{ir}).
- (b) The left-turning tangential CC turn on the CC circle, centred at Ω_{tl} , which is μ -tangential to both the right CC circle at $\mathbf{q}_{\text{start}}$ and the right CC circle at \mathbf{q}_{end} .
- (c) The right-turning CC turn at the end configuration, \mathbf{q}_{end} (i.e the one on the CC circle centred at Ω_{gr}).

With the above-mentioned geometric construction, the *LRL* and *RLL* paths can be computed. The computed paths can then be sorted according to path length. These are the two solution paths in the case where the Euclidean distance between $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} is less than $2r_{\text{CCT}}$.

We now proceed to the computation of the continuous-curvature local paths in the case where the separation between the initial and goal CC circles is greater than $2r_{\text{CCT}}$, such as the example given in Figure 4.12. In this case, the tangential curves between each initial and goal circle pair are straight lines. Since there are 4 circles, we again have 16 possible tangent lines, but only 4 result in feasible paths for a vehicle that can only move forward as in the Dubins case. As earlier stated, the tangency relationship we are interested in is μ -tangency rather than the classical tangency and it is depicted in Figure 4.13 for the possible paths.

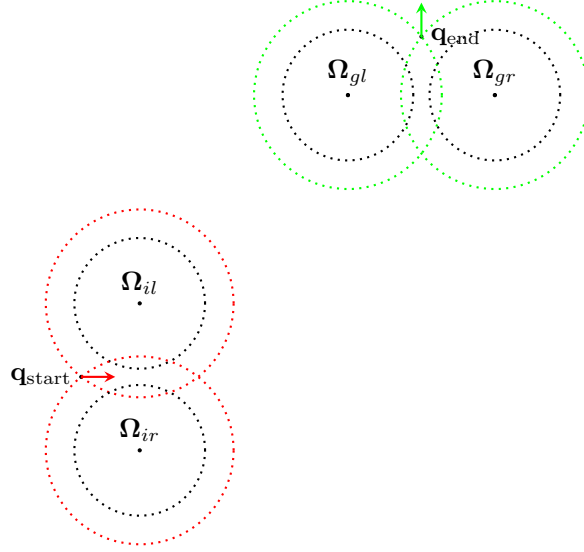


Figure 4.12: Turning circles for a continuous-curvature local path-planning query with $\mathbf{q}_{\text{start}} = [0 \ 0 \ 0]^T$ and $\mathbf{q}_{\text{end}} = [10 \ 10 \ \frac{\pi}{2}]^T$. The initial configuration is marked with a red arrow representing its position and orientation. A green arrow is used to show the same attributes for the end configuration. The turning circles at each configuration are marked with corresponding colours. Once these circles are computed, the remaining part off the planning process is to connect them with μ -tangent lines and then compute CC turns at each turning circle.

From the placement of μ -tangent lines for each of the *LSL*, *LSR*, *RSL* and *RSR* cases in Figure 4.13, the μ -tangential configurations $\mathbf{q}_1 = [x_1 \ y_1 \ \theta_1]^T$ and $\mathbf{q}_2 = [x_2 \ y_2 \ \theta_2]^T$ can be determined. In each case, the configuration \mathbf{q}_1 marks the end of the first CC turn, while \mathbf{q}_2 marks the start of the second CC turn. By using these configurations in conjunction with $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} , the deflection required for each turn can be computed. Particularly, the deflection required at the first turn is $\delta = \theta_1 - \theta_{\text{start}}$, while that required at the second turn is given by $\delta = \theta_{\text{end}} - \theta_2$. From these values of the required deflection, the constituting manoeuvres of each CC turn can be determined and concatenated to form that CC turn. Each continuous-curvature *CSC* path is then a concatenation of the first CC turn, the μ -tangent line segment and the second CC turn in the specified order.

With this described geometric construction, the *LSL*, *LSR*, *RSL* and *RSR* paths can be computed. The computed paths can again be sorted according to path length. This is the set of solution paths in the case where the Euclidean distance between $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} is greater than $2r_{\text{CCT}}$. It is important to point out that, as stated earlier during the discussion on CC turns, some of these paths may contain a self-intersecting turn of the second case (i.e. $\delta_{\text{min}} + \pi \leq \delta < 2\pi$). However, as pointed out in that discussion, when this happens, at least one of the paths among the four will not have such a self-intersection. Since the paths are sorted according to path length, the loop-less path will always be given priority over the rest.

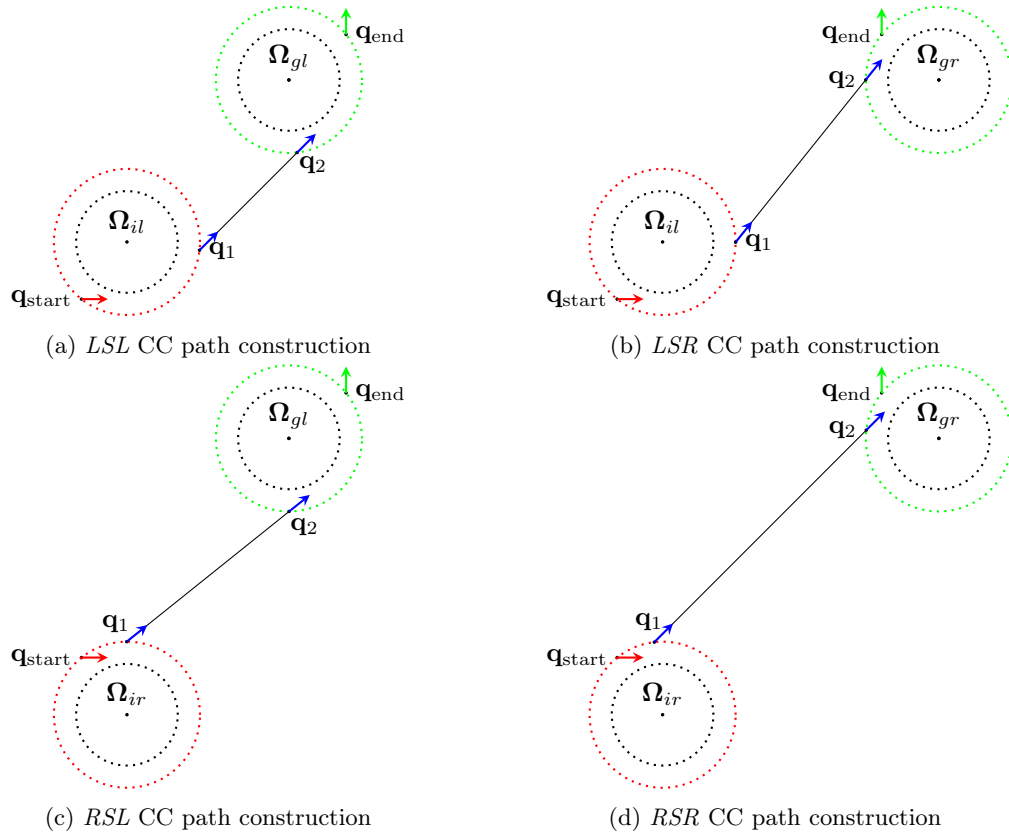


Figure 4.13: Depiction of μ -tangent placement for connecting the start-end CC circle pairs of each of the possible continuous-curvature *CSC* paths. The points of tangency are labelled \mathbf{q}_1 and \mathbf{q}_2 respectively for the start and end turning circles. These points, together with the angle of the μ -tangent, form the end configuration of the first turn and the start configuration of the second turn. The CC turn at the start can thus be computed by using the value of the deflection between $\mathbf{q}_{\text{start}}$ and \mathbf{q}_1 . Likewise, the CC turn at the end configuration can be computed using the value of the deflection between \mathbf{q}_2 and \mathbf{q}_{end} . It is important to note that while the self-intersecting turns of the second case ($\delta_{\min} + \pi \leq \delta < 2\pi$) exist in other paths (b, c, d), a path without self-intersection (a) always exists and it is given priority over self-intersecting paths.

At this point, the computation of *CCC* (i.e. *LRL* and *RLR*) and *CSC* (i.e. *LSL*, *LSR*, *RSR* and *RSL*) continuous-curvature paths for local path-planning queries with standard inputs has been described geometrically – this is the CC LPM with standard inputs. An implementation of the developed LPM is achieved by implementing this geometric construction process through a computer algorithm. Again, the developed LPM has been written in MATLAB and C++ for the previously stated reasons. In the next subsection, we present results obtained from runs of the developed LPM.

4.2.3.2.4 Results from Runs of the Developed CC LPM with Standard Inputs

The previous subsection presented the development of the CC LPM with standard inputs by adapting the Dubins LPM with standard inputs for curvature continuity. We now show outputs generated by our MATLAB implementation for given local path-planning queries to demonstrate the working of the algorithm.

We begin by demonstrating the developed algorithm through a local path-planning query, which results in a path that is purely made up of turns. Figure 4.14 shows the continuous-curvature *LRL* and *RLR* paths generated by our MATLAB implementation of the developed continuous-curvature local path planner with standard inputs for such a local path-planning query.

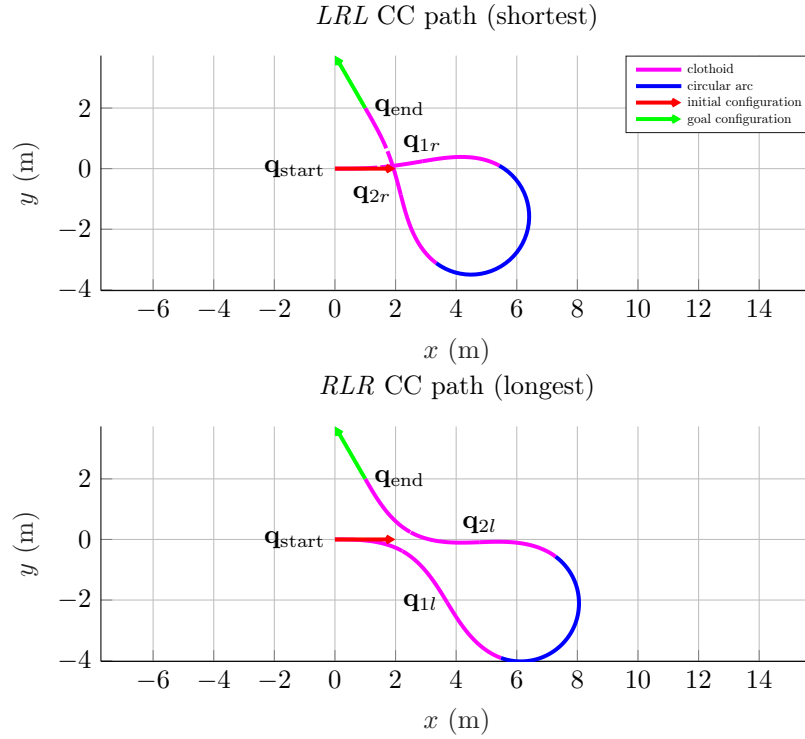


Figure 4.14: Continuous-curvature paths generated for a local path-planning query with $\mathbf{q}_{\text{start}} = [0 \ 0 \ 0]^T$ and $\mathbf{q}_{\text{end}} = [1 \ 2 \ \frac{2\pi}{3}]^T$. The start and end configuration of each path are $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} respectively. For the *LRL* path, \mathbf{q}_{1r} and \mathbf{q}_{2r} respectively mark the start and end of the second CC turn – they are also respectively the end and start configurations of the first and last turns. \mathbf{q}_{1l} and \mathbf{q}_{2l} mark similar points on the *RLR* path. The paths are sorted according to path length as shown, with the *LRL* path being the shortest.

For a local path-planning query that results in *CSC* paths, the resulting paths as generated by a run of our MATLAB implementation of the developed continuous-curvature local path planner with standard inputs are shown in Figure 4.15, with the paths sorted according to path length.

The presented plots for planned paths demonstrate the working of the development of the continuous-curvature LPM of the form $\text{LPM}([x_{\text{start}} \ y_{\text{start}} \ \theta_{\text{start}}]^T, [x_{\text{end}} \ y_{\text{end}} \ \theta_{\text{end}}]^T)$. This LPM was developed by adapting its Dubins counterpart for curvature continuity. In the next section we proceed to the development of the second version of the LPM, which treats the orientation at the goal configuration as a “don’t care” condition.

4.2.3.3 Continuous-curvature LPM with “Don’t Care” Goal Orientation

In this subsection we go through the same process as that followed in the previous subsection, with the difference being that the local path planner we develop in this subsection does not require the orientation at \mathbf{q}_{end} to be specified. In other words, the local path planner treats this orientation as a “don’t care condition”. This local path planner has a function prototype or header of the form: $\text{LPM}([x_{\text{start}} \ y_{\text{start}} \ \theta_{\text{start}}]^T, [x_{\text{end}} \ y_{\text{end}}]^T)$. The local path planner is developed such that it generates paths with a continuous-curvature profile. Again, we follow the same approach to developing a continuous-curvature LPM as that used in the previous subsection wherein we start by developing a discontinuous-curvature one based on Dubins’ formulation, and then adapt the resulting LPM for curvature continuity. The difference in this case is that since the Dubins LPM requires both $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} to be specified in both position and orientation, we need to first adapt Dubins LPM so that the orientation at \mathbf{q}_{end} is treated as a “don’t care” condition, before adapting it for curvature continuity. The following subsection focuses on the development of Dubins LPM adapted for the

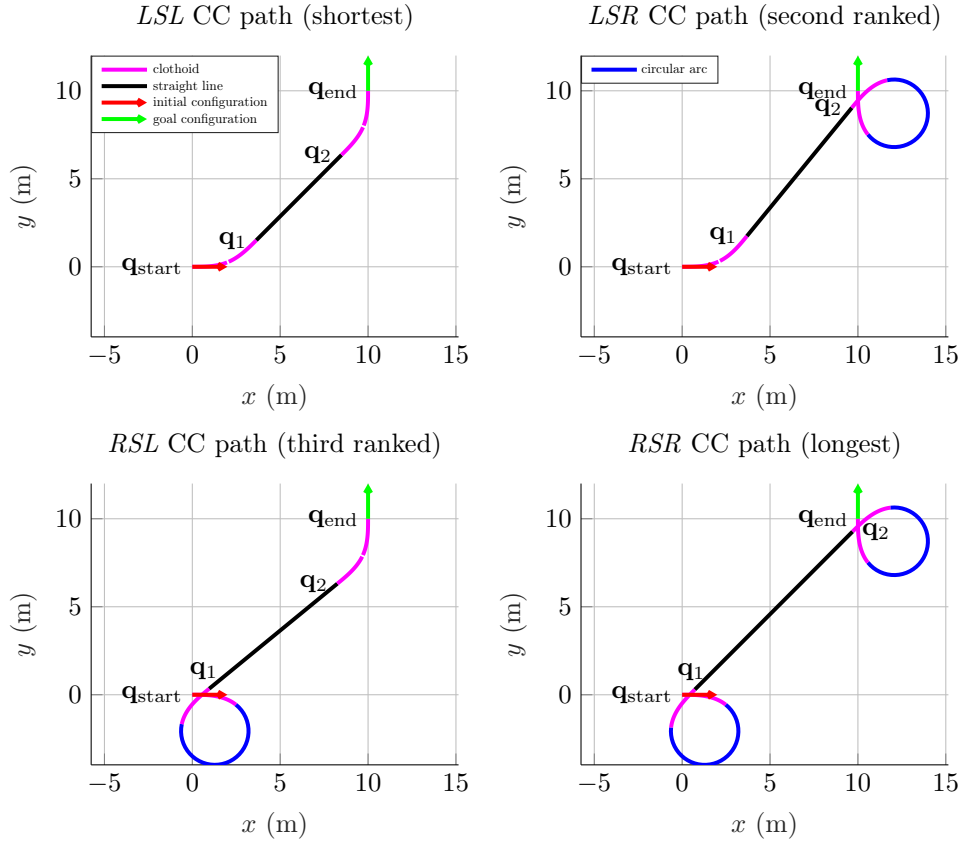


Figure 4.15: Continuous-curvature *CSC* paths generated by our continuous-curvature local path-planning algorithm for a local path-planning query with $\mathbf{q}_{\text{start}} = [0 \ 0 \ 0]^T$ and $\mathbf{q}_{\text{end}} = [10 \ 10 \ \frac{\pi}{2}]^T$. The start and end configuration of each local path are labelled $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} respectively.

“don’t care” orientation at \mathbf{q}_{end} . The developed LPM is then adapted for curvature continuity in Subsection 4.2.3.3.3, which follows thereafter.

4.2.3.3.1 Development of Dubins LPM with “Don’t Care” Goal Orientation

We now present the development of a variant of Dubins LPM that treats the orientation at \mathbf{q}_{end} as a “don’t care” condition. In the original Dubins local path-planning method [107], the orientation at both $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} is a constraint that has to be satisfied. A modification to the original formulation is therefore required and it is the subject of this subsection.

A fundamental difference between this variant of the Dubins LPM and the original formulation is that the paths computed by this variant constitute only two manoeuvres as compared to the three manoeuvres in the original case. These two manoeuvres are an initial circular-turn manoeuvre and a straight-line manoeuvre by which \mathbf{q}_{end} is reached. This leads to only two possible paths between $\mathbf{q}_{\text{start}}$ and \mathbf{q}_{end} , which are respectively denoted *LS* and *RS* – a notation that is consistent with Dubins’ original notation. The *LS* path corresponds to a left turn from $\mathbf{q}_{\text{start}}$ followed by a straight-line manoeuvre that reaches \mathbf{q}_{end} . In a similar manner, the *RS* path corresponds to a right turn from $\mathbf{q}_{\text{start}}$ followed by a straight-line manoeuvre that reaches \mathbf{q}_{end} . A much richer notation that embeds the length of each manoeuvre is: $L_{\alpha}S_d$ and $R_{\alpha}S_d$. This notation is summarised in Table 4.3.

Computationally, this variant of Dubins LPM is significantly cheaper than its original counterpart as only one pair of turning circles (i.e the ones at $\mathbf{q}_{\text{start}}$) needs to be computed, and the significant drop of the number of possible paths from six to two eases the computational burden. Figure 4.16 depicts the geometric construction used to compute solution paths for Dubins LPM with “don’t care” goal orientation, which is explained next.

Path type	Manoeuvre 1	Manoeuvre 2
RS	R_α	S_d
LS	L_α	S_d

Table 4.3: Possible paths for Dubins LPM with “don’t care” orientation and their manoeuvre specifications. The subscript, α , specifies the angular length of the turning manoeuvre and the subscript, d , denotes the length of the straight-line manoeuvre.

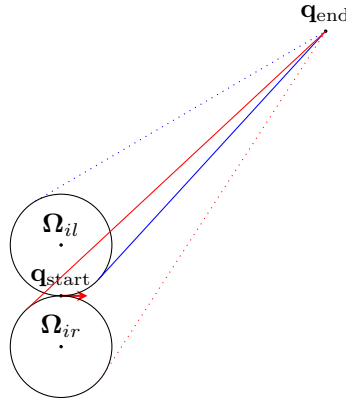


Figure 4.16: Tangent placement for the construction of the adapted Dubins CS paths in which the goal orientation is treated as a “don’t care” condition. The turning circles at the start configuration are the same as before. Each turning circle has two tangent lines that connect it \mathbf{q}_{end} . One of these tangents (dotted) results in the vehicle reaching the end configuration in reverse, while through the other tangent (solid) the end configuration is reached in forward motion. For each turn, only the path with a tangent resulting in forward motion is considered for path construction. For ease of distinction, different colours are used for the different turns, with red for the right turn and blue for the left turn.

The computation begins with the construction of the pair of turning circles at \mathbf{q}_{start} , as described in the original case. Once these circles have been determined, the next step is to determine the required straight-line segments. The defining property of these lines is that they must be tangential to the turning circle and also pass through \mathbf{q}_{end} . The tangent points for each turning circle can be computed geometrically by constructing a circle centred at \mathbf{q}_{end} with radius equal to the Euclidean distance from the corresponding turning circle’s centre, Ω , to \mathbf{q}_{end} , $\|\Omega - \mathbf{q}_{end}\|$. For each turning circle, there are two such tangent points; however, only one results in a feasible path for a vehicle moving forward. The other path does not belong to the Dubins class, as it would require a change to reverse motion at the tangent point and as such \mathbf{q}_{end} can be reached in reverse. The four possible tangents are shown in Figure 4.16 with the infeasible ones dotted while the feasible ones are solid. From this figure it is clear that the orientation by which each \mathbf{q}_{end} is reached is indeed different for each tangent line. The construction of the resulting paths is shown in Figure 4.17, with the angles of each turn and the length of each straight line segment marked.

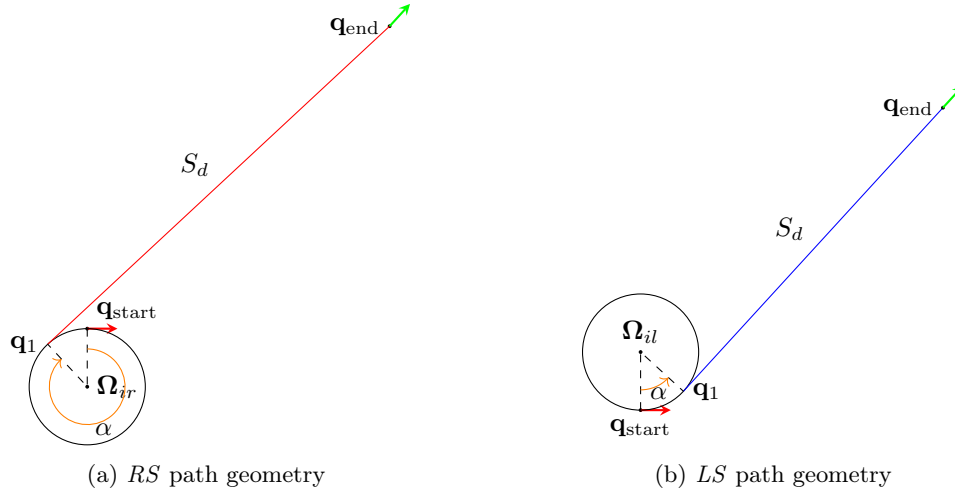


Figure 4.17: Geometric construction of Dubins *LS* and *RS* paths. For each turn the tangent resulting in forward motion is extracted from Figure 4.16. The angle, α , represents the angular length of each turn and \mathbf{q}_1 is the point of tangency. S_d is then the length of the straight-line manoeuvre. It is important to note that the orientation at \mathbf{q}_{end} is unknown at the time of computing the paths (see Figure 4.16), it only becomes known once the paths have been computed, as shown in this figure, and for each path it is the same as the orientation of line segment of that path.

The geometric construction of solution paths for the Dubins LPM with “don’t care” goal orientation has now been described. By implementing this geometric process in a computer algorithm, an implementation of the developed LPM is achieved. The developed LPM has again been written in MATLAB and C++. In the next subsection, we present results obtained from sample runs of the developed LPM.

4.2.3.3.2 Results from Runs of Developed Dubins LPM with “Don’t Care” Goal Orientation

In the previous subsection, the Dubins LPM with “don’t care” goal orientation has been developed. In this subsection, we conclude our discussion on this LPM by presenting results from sample runs for a given local path-planning query. Figure 4.18 shows plots of the paths generated by our MATLAB implementation of the LPM for the local path-planning query.

We have now developed a local path planner that is capable of handling a local path-planning query in which the orientation at the goal configuration is unspecified and as such, treated as a “don’t care” condition. This local path planner produces paths which have a discontinuous-curvature profile just like the original Dubins LPM. The next step is to adapt this LPM to the continuous-curvature case and is discussed next.

4.2.3.3.3 Development of the Continuous-curvature LPM with “Don’t Care” Goal Orientation

The preceding subsection discussed the development of a Dubins-type local path planner capable of solving local path-planning queries in which the orientation at \mathbf{q}_{end} is treated as a “don’t care” condition, i.e. queries of the form $\text{LPM}([x_{\text{start}} \ y_{\text{start}} \ \theta_{\text{start}}]^T, [x_{\text{end}} \ y_{\text{end}}]^T)$. The developed LPM however suffers from the discontinuous curvature phenomena associated with Dubins paths. We now adapt this LPM for curvature continuity. This is achieved in a similar manner to the one used to adapt the discontinuous-curvature paths in the case of the LPM with standard inputs, i.e. the one with prototype: $\text{LPM}([x_{\text{start}} \ y_{\text{start}} \ \theta_{\text{start}}]^T, [x_{\text{end}} \ y_{\text{end}} \ \theta_{\text{end}}]^T)$, to their continuous-curvature version.

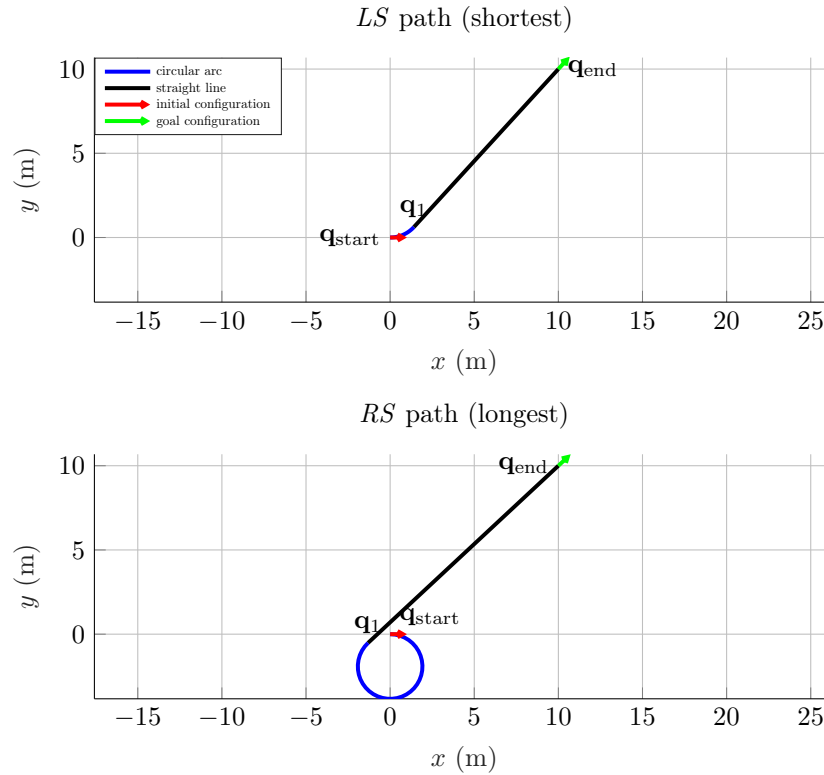


Figure 4.18: Solution paths generated by our MATLAB implementation of the adapted Dubins LPM for a local path-planning query with $\mathbf{q}_{\text{start}} = [0 \ 0 \ 0]^T$ and $\mathbf{q}_{\text{end}} = [10 \ 10]^T$. The algorithm sorts the paths according to path length as shown, with the *LS* path being the shortest in this case.

The construction process is shown in Figure 4.19. It begins by the replacement of Dubins turning circles at the initial configuration with CC turning circles and the subsequent determination of μ -tangent lines in place of the classical tangent lines in Figure 4.16. Tangents that result in paths that would require the vehicle to switch to reverse motion at the tangent point are again deemed infeasible since they do not belong to the Dubins class.

With the μ -tangents placed, the *LS* and *RS* paths can be respectively computed through the geometric construction shown in Figures 4.20(a) and 4.20(b). Each of these paths consists of a CC turn followed by a straight-line manoeuvre. The CC turn can be computed by first using the μ -tangential configuration, \mathbf{q}_1 , in conjunction with $\mathbf{q}_{\text{start}}$ to determine the required deflection, which is given by $\delta = \theta_1 - \theta_{\text{start}}$. With the deflection computed, its value can be used to determine the nature of the CC turn and hence the constituting manoeuvres according to the criteria outlined under the discussion of CC turns in Subsection 4.2.3.2. The CC turn is then a concatenation of these manoeuvres. The straight-line manoeuvre is simply the straight line from the μ -tangential configuration, \mathbf{q}_1 , to \mathbf{q}_{end} . The resulting path is then a concatenation of the CC turn and the straight-line manoeuvre.

The geometric process described above has been implemented in MATLAB and C++, resulting in an implementation of the developed CC LPM with “don’t care” goal orientation. The next subsection concludes the discussion on this LPM by presenting results from sample runs.

4.2.3.3.4 Results from Runs of the Developed CC LPM with “Don’t Care” Goal Configuration

The preceding subsection has presented the development of the CC LPM with “don’t care” goal orientation through the adaptation of Dubins LPM with “don’t care” goal orientation for curvature continuity. The developed LPM has been written in MATLAB and C++. This subsection

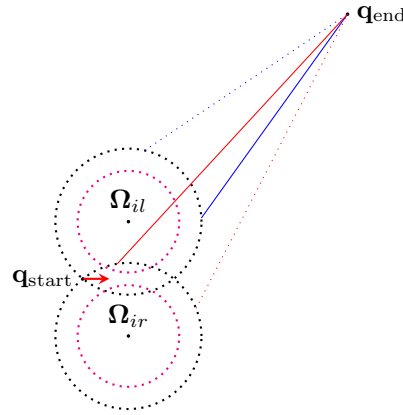


Figure 4.19: Depiction of geometric construction of continuous-curvature paths, with the goal orientation treated as a “don’t care” condition. Each CC circle has two μ -tangent lines that connect it to \mathbf{q}_{end} . One of these μ -tangents (dotted) results in the vehicle reaching \mathbf{q}_{end} in reverse, while the other μ -tangent (solid) results in \mathbf{q}_{end} being reached in forward motion. For each CC turn, only the path with a tangent resulting in forward motion is considered for path construction. For ease of distinction, different colours are used for the different turns, with red for the right turn and blue for the left turn.

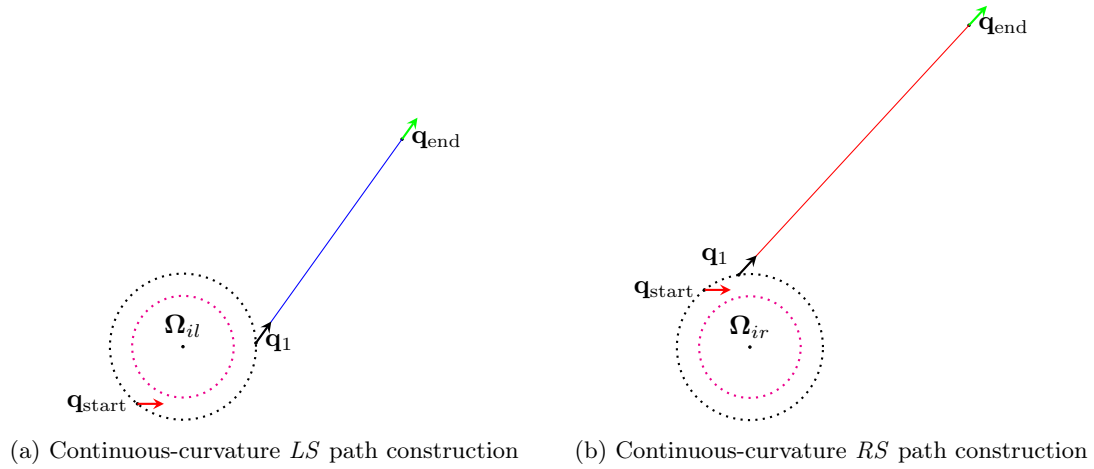


Figure 4.20: Depiction of μ -tangent placement for the construction of *LS* and *RS* continuous-curvature paths. In each case, \mathbf{q}_1 marks the end of the CC turn. The position component of this configuration is given by the point of intersection between the CC circle and the μ -tangent, while its orientation component is simply the angle of the μ -tangent. The resulting CC turn can be computed by first calculating the deflection between $\mathbf{q}_{\text{start}}$ and \mathbf{q}_1 and determining the constituting manoeuvres of the CC turn using the value of the deflection. The resulting path is a concatenation of the CC turn and the μ -tangent line.

demonstrates the working of the LPM by presenting planned paths for a given local path-planning query. These solution paths are shown in Figure 4.21 where they are sorted according to path length. Given any local path-planning query of the form: $\text{LPM}([x_{\text{start}} \ y_{\text{start}} \ \theta_{\text{start}}]^T, [x_{\text{end}} \ y_{\text{end}}]^T)$, the LPM computes the *LS* and *RS* solution paths and sorts them according to path length.

At this point, we have developed a local path planner capable of planning a continuous-curvature path given a local path-planning query in which the orientation at \mathbf{q}_{end} is unspecified and as such is treated as a “don’t care” condition. The developed LPM has been demonstrated through plots

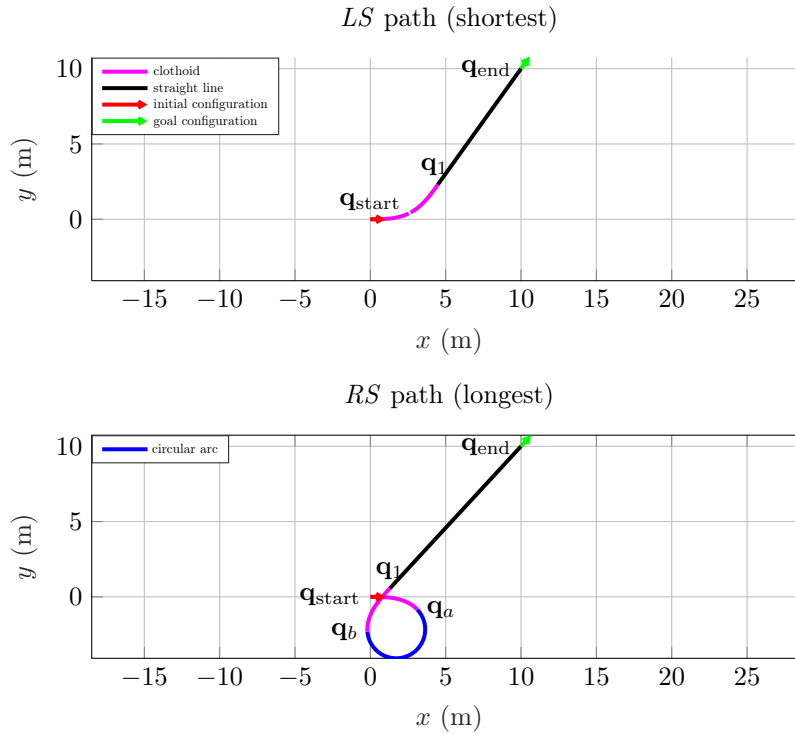


Figure 4.21: Solution paths generated by our MATLAB implementation of the adapted CC LPM with “don’t care” orientation for a local path-planning query with $q_{start} = [0 \ 0 \ 0]^T$ and $q_{end} = [10 \ 10]^T$. The algorithm sorts the paths according to path length as shown, with the *LS* path being the shortest in this case.

of planned paths generated by our MATLAB implementation. This is the second version of the continuous-curvature LPM developed in this thesis, with the first one (developed in Section 4.2.3.2) requiring both q_{start} and q_{end} to be fully specified in both position and orientation. As such, this marks the end of our local path planner development. The next section summarises the work done in this chapter.

4.2.4 Local Path-planning Method Summary

In this chapter, we have successfully developed two versions of the local path-planning algorithm. One version of the local path planner requires q_{start} to be specified as a “position and orientation” vector while q_{end} is specified as a “position only” vector. We chose to call it the *continuous-curvature local path planner with “don’t care” goal orientation*, shortened: CC LPM with “don’t care” goal orientation. The second one requires both q_{start} and q_{end} to be specified as “position and orientation” vectors. We referred to it as the *continuous-curvature local path planner with standard inputs*, shortened: CC LPM with standard inputs. The function prototypes for these local path planner versions are respectively: $LPM([x_{start} \ y_{start} \ \theta_{start}]^T, [x_{end} \ y_{end}]^T)$ and $LPM([x_{start} \ y_{start} \ \theta_{start}]^T, [x_{end} \ y_{end} \ \theta_{end}]^T)$. They return a set of feasible local paths for a given LPM query, with each path given as a sequence of geometric motion primitives to facilitate collision detection when used within any global path planner. The working of the developed algorithms has been demonstrated through plots of planned paths generated by our MATLAB implementations for given local path-planning queries. The results of this section will be used in the development of the optimised global path planner which is the subject of the next Chapter.

The developed algorithms have been written in MATLAB and C++. MATLAB has been used for fast prototyping and testing of each developed algorithm. The translation to C++ has been done to make it possible to deploy the developed algorithms in the Robot Operating System (ROS)

during integration with other constituting modules of the overall autonomous navigation system, which is the subject of Part III.

Chapter 5

Development of Optimised Global Path Planning Algorithms

As explained in a number of earlier portions of this thesis, including Sections 1.2, 2.2 and 4.1, the global path-planning algorithm can be viewed as a higher-level planner that makes use of calls to a low-level planner known as the local path planner, which has been developed in Chapter 4, and a collision detector to compute a feasible collision-free path in the configuration space, \mathcal{C} , from the robot's initial configuration, \mathbf{q}_I , to a desired goal configuration, \mathbf{q}_G , in a given environment. Thus the inputs to the algorithm are:

1. a map of the environment,
2. the initial configuration, and
3. the desired goal configuration.

The map of the environment is used for collision detection purposes – in collision detection, the collision detector uses the map of the environment to determine if a computed portion of the path from \mathbf{q}_I to \mathbf{q}_G is free from collisions. The output of the global path planner is a collision-free path from \mathbf{q}_I to \mathbf{q}_G . Programmatically, the global path planner can be viewed as an algorithm taking a map and a configuration pair $(\mathbf{q}_I, \mathbf{q}_G)$ as input and returning a path (a closely spaced sequence of configurations) connecting this configuration pair. In this sense, the algorithm can be thought of as a function with function prototype: `path globalPlan(map, \mathbf{q}_I , \mathbf{q}_G)`. Both the initial and goal configuration are “position and orientation” vectors, i.e. $\mathbf{q}_I = [x_I \ y_I \ \theta_I]^T$ and $\mathbf{q}_G = [x_G \ y_G \ \theta_G]^T$. The body of this function is a sequence of computational steps involved in finding the solution path, and these steps depend on the technique chosen to solve the path planning problem. An optimised global path planner goes an extra mile – instead of simply finding a feasible path, it strives to find one with the least cost. In this project, the cost of a path is measured by its length.

The review of existing techniques for optimised global path planning was the subject of Chapter 2. It is through this review that the optimised global path-planning approach employed in this project was selected. Section 5.2 reintroduces the selected approach by reflecting on the key take-home points from Chapter 2, which led to its selection. After reintroducing the path-planning approach, the section concludes with the organisation of the rest of the chapter by pointing out sections in which each of the components of the planning approach are developed. Before reintroducing the path-planning approach, we begin by considering how the inputs to our global path planning algorithm will be obtained.

5.1 Acquiring Inputs for the Global Path Planner

The three inputs to the global path-planning algorithm are: a map of the environment, the initial configuration and the goal configuration. In this section we consider how these inputs will be acquired. The desired goal configuration can be easily obtained by allowing the user to enter it by means of a GUI or keyboard input. More generally, the goal configuration can be obtained from a human user or it can be determined by an artificial decision-making agent within which the global

path planner is used. In this project, the goal configuration is acquired by means of a GUI that will be seen in Section 7.5. The initial configuration can also be obtained manually through the same methods for the path planner, with the robot picked and placed at this location for plan execution after planning is complete. Alternatively, the initial configuration can be obtained automatically from the localisation module of the AutoNav framework. As stated in the definition of this project's scope, the implementation of the localisation module is outside the scope of this project. As such, for purposes of this project, the existence of a localisation module will be assumed, and the initial configuration manually given to the path planner.

The remaining input is the map of the environment. This map can be acquired in a number of ways. One way would be to manually inspect the environment in which the robot is required to operate, note obstacle locations and then represent this information in a digital format that can be fed to the algorithm. In a fully autonomous system, this function is not done manually, but is rather automated through the mapping module of the AutoNav framework, which uses sensor observations to build a map of the environment. Like the localisation module, the mapping module is outside the scope of this project. As a result, an existing map will be uploaded and used in our autonomous navigation system. Once a future project implements a working mapping module, it can be used in place of the pre-loaded map with minimal or no modifications, given that it has the same format as the pre-loaded map. In this project it thus suffices to discuss commonly-used map representations in robotics and select a convenient one for our application. This is the subject of the next subsection.

5.1.1 Robotic Map Representations: Selecting a Map Representation

Before beginning a discussion of the different map representations, we start by defining what a map is. A map, m , is defined as a list of properties of objects in the environment:

$$m = \{m_1, m_2, \dots, m_N\}, \quad (5.1)$$

where N is the total number of objects in the environment, and each m_n , with $n \leq N$, specifies a property [161]. There are two classical categories of maps, namely metric and topological maps [23]. They are discussed next.

Metric Maps: Metric maps represent the environment by storing the positions of objects in the environment in a common reference frame. These positions can be stored in the form of points, lines, or other geometric shapes with their coordinates in a 2D space. Since the coordinates of objects are represented in a common reference frame in metric maps, the coordinates make it possible to infer distances between objects as well as the distance of an object from any point in the reference frame.

Topological Maps: Unlike metric maps, topological maps represent the environment as a set of distinct places and the way a robot can go from one place to another. In other words, a topological map can be viewed as a graph-like structure in which the graph vertices or nodes are the distinct places that can be reached and the edges denote the fact that it is possible to go from one place to a neighbouring place.

Figure 5.1 shows an example of a real environment (top) with both metric and topological information. This environment is then decomposed to a metric map (bottom left) and a topological map (bottom right). Here, the metric components are the points with given coordinates A and B as well as the solid straight lines that are at respective coordinates in the 2D space. The start and goal configurations (A and B) are specified by their specific coordinates. The topological components are the places R, C, D, T (where $R = \text{room}$, $C = \text{corridor}$, $D = \text{door}$ and $T = \text{turn}$), which are not necessarily at specific coordinates in the 2D space. Unlike in the metric case, the initial and goal configurations in the topological case are each specified as being within a given place in the topological map rather than being specified by their coordinates.

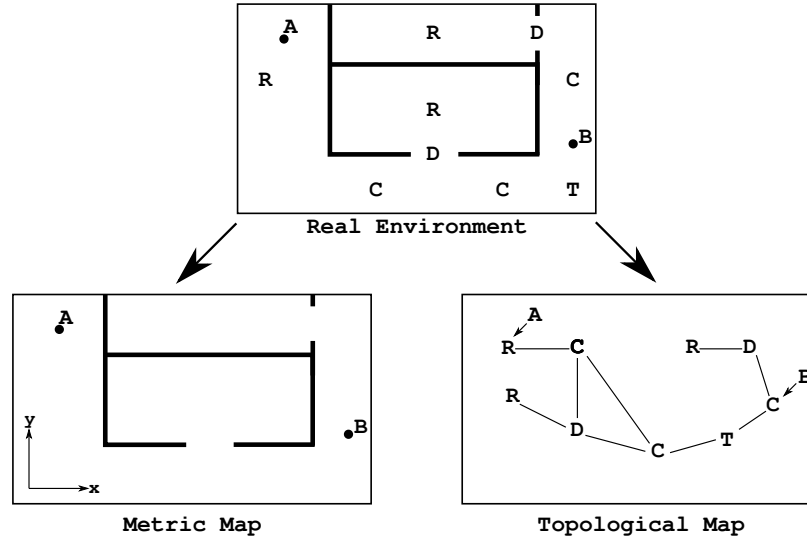


Figure 5.1: An illustration of the difference between metric and topological maps (adapted from Filliat et al. [23]). As shown metric maps represent locations and objects in the map with coordinates in a common reference frame. Topological maps on the other hand only capture places of interest in the environment and the routes between these places.

Comparison Between Metric and Topological Maps

From the presented high-level overview of metric and topological maps, it is clear that metric maps can be used to represent the configuration space, \mathcal{C} , in which the robot operates and that the metric information makes it possible to determine whether any randomly chosen configuration lies on an obstacle region (\mathcal{C}_{obs}) or in free space ($\mathcal{C}_{\text{free}}$). This is an important fundamental requirement for sampling-based path-planning algorithms. Topological maps, on the other hand, only capture connections between places without necessarily capturing the configuration space of the robot – for this reason, the previously stated fundamental requirement for sampling-based path-planning algorithms is not fulfilled in the topological case. Therefore, metric maps are well suited for sampling-based path planning, which is the context of our project, and topological maps are not. Our choice for map representation is thus narrowed down to the different kinds of metric maps, and choosing an appropriate type of metric map for our application. This is the subject of the remainder of this section.

Types of Metric Maps

Different types of metric maps exist. They differ in the way in which they represent the environment. There are two ways in which the environment can be represented in metric maps, namely *feature representation* and *free-space representation*. In feature representation, the metric map explicitly stores features that are perceived by the robot's sensors, along with their positions. Such features could be lines defining polygonal boundaries of obstacles. These lines are typically extracted from point clouds detected by the robot's sensors. An example of such a map, known as a *feature map* [23], is shown in Figure 5.2(a) for the environment given in Figure 5.1. The same metric map can be expressed through free-space representation, where the portion of the environment that is accessible to the robot is marked. The mostly used approach for this representation is the *occupancy grid map* [23, 161], which is shown in Figure 5.2(b) for the same example environment shown in Figure 5.1. In this case, the environment is discretised into a regular high-resolution grid as shown and each cell is assigned with a probability of being occupied by an obstacle.

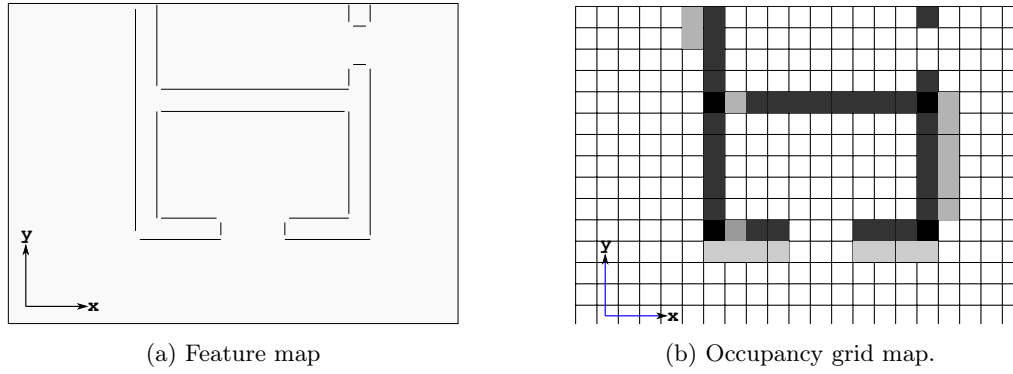


Figure 5.2: The two ways in which a metric map can be expressed. In this example, the original environment is shown in Figure 5.1. In the occupancy grid map, the intensity of the colour assigned to each cell is proportional to the probability of its occupancy. White cells are unoccupied and black cells are certainly occupied.

Selection of an Appropriate Metric Map

Selecting between either feature maps or occupancy grid maps for our application requires an understanding of the operations that will be performed on the map. In the context of a sampling-based global path planner, the map is used in the collision-detection step. This step performs distance computations to obstacles in the map for each generated candidate local path in each iteration of the algorithm, to decide whether to add that candidate local path as a new branch of the planning tree. According to LaValle [25], this is a critical component of sampling-based path planning and it is the step where the majority of computation time is spent in a typical sampling-based path-planning algorithm. As such, it is important to select a map representation that aids efficiency of collision detection. With the collision detection requirement of our path-planning approach in mind, it is clear that an occupancy grid map would incur a higher computational cost if directly used for this purpose. This is because, for collision detection, there would be many cells along certain paths, so the collision tests would be expensive. In contrast, feature maps store the obstacle information in a compact way, using geometric primitives such as the straight lines in Figure 5.1. Other compact forms involve representing obstacles as polygons. This can be achieved by grouping lines representing obstacle edges into polygons. Due to their compactness in comparison to occupancy grid maps, which aids efficiency of collision detection, feature maps are chosen for our application.

A good observation is that if the occupied cells of the occupancy grid map can be grouped and converted into geometric primitives such as lines, polygons and circles, then the result will be equivalent to a feature map. Therefore, even though occupancy grid maps have not been selected for map representation in our application, it is possible to convert them into feature maps which have been chosen as suitable.

With the purpose of the optimised global path planner introduced and methods for acquiring algorithm inputs, namely the map of the environment and the initial and goal configurations discussed, we now proceed to reintroduce the particular optimised global path-planning approach that was selected for use in this project.

5.2 The Optimised Global Path-planning Approach

As concluded in Subsection 2.6.6, the optimised global path-planning approach used in this project combines selected sampling-based path-planning algorithms with path optimisation for accelerated convergence towards the optimal solution. This section reintroduces this path-planning approach. The reintroduction of the path-planning approach helps to put the remaining sections of this chapter in context.

The selected sampling-based path planning algorithms are two variants of the rapidly-exploring random tree (RRT), namely the anytime RRT and the informed RRT*. These two algorithms work in a similar way; they both strive towards the optimal solution by performing successive searches in the configuration space, \mathcal{C} , where the sampled subset of \mathcal{C} is successively reduced in each new search by bounding \mathcal{C} using the cost of the path found in the previous search. The difference is in the manner in which each of them grows the search tree. The anytime RRT has two extremes. In one of these extremes, it considers only one existing tree vertex when extending the tree towards a sample – the one that is closest to the sample. Under this setting, the anytime RRT extends the search tree exactly the same way as the basic RRT, which is known to be quick and efficient, though producing paths that are almost-surely suboptimal. In the second extreme, when extending the search tree towards a sample, the anytime RRT attempts connections from multiple existing tree nodes and chooses the cheapest among these. As a result of attempting multiple connections before choosing one, the anytime RRT is generally slower in this second extreme than in the first; however, it produces paths of better quality than in the first extreme since it chooses low-cost extensions. Just like the anytime RRT's second extreme, the informed RRT* also attempts multiple extensions and chooses the cheapest among them when extending the tree towards a sample. However, it performs an additional step known as tree rewiring after a successful addition of a new node. In this step, the newly-added node is considered as a replacement parent for each neighbouring node, if it results in a cheaper path to that neighbour than the existing path that goes via the neighbour's current parent. Due to this additional step, the informed RRT* is generally slower than both extremes of the anytime RRT; however, due to the rewiring process, it produces paths of better quality than both extremes of the anytime RRT.

Our optimised path-planning approach works by combining the selected path planners introduced above with path optimisation. The role of path optimisation in the approach is to reduce the cost of each solution found by each of the selected path planners before being used to bound the search space in the next search, accelerating the rate of convergence of the path planners. What makes it more interesting to consider the application of the path-optimisation step to accelerate the convergence of the selected path planners is the fact that the main difference between the anytime RRT and the informed RRT* is in the quality of solutions they produce and the amount of time they take to compute those paths. Particularly, in its first extreme, the anytime RRT quickly produces paths that are said to be almost-surely suboptimal. Paths produced in the second extreme of the anytime RRT are also almost-surely suboptimal, but are cheaper than those produced in the first extreme; however, they are produced in increased computation time. The informed RRT* is asymptotically optimal – it is guaranteed to find the optimal path with the number of iterations approaching *infinity*. For the anytime RRT, path optimisation is the only hope if optimality is to be attained, since on its own, the anytime RRT is almost-surely suboptimal. On the other hand, for the informed RRT*, path optimisation is used to accelerate the algorithm's convergence so that it can quickly converge.

With a path-optimisation step incorporated into the anytime RRT and the informed RRT*, it becomes interesting to investigate the effectiveness of the various path-optimisation algorithms, selected in Section 2.8.3, in accelerating the convergence of the path planners. Of particular importance is to find out if the application of the path-optimisation step can help a quick, almost-surely suboptimal path planner, such as the first extreme of the anytime RRT, to attain comparable or better performance than an asymptotically optimal path planner, like the informed RRT*.

It is important to reiterate that both the path-planning algorithms and the path-optimisation algorithms should produce curvature-continuous paths, so that they are executable by the vehicles targeted by the thesis. With the path-planning approach reintroduced, we now summarise the adaptations and extensions in point form to facilitate the mapping out of appropriate sections of this chapter, in which each component of the path-planning approach is developed. The adaptations and extensions can be summarised as follows:

1. Adaptation of the basic RRT, anytime RRT and informed RRT* algorithms for continuous-curvature path planning.
2. Development of continuous-curvature path-optimisation algorithms.

3. Incorporating developed path-optimisation algorithms into the benchmark path planners developed in the first point.
4. Analysing the effectiveness of each path-optimisation in accelerating the convergence of each benchmark path-planning algorithm.

The adaptation of the RRT, anytime RRT and informed RRT* algorithms for continuous-curvature path planning is the subject of Section 5.4. This adaptation requires a local path-planning method (LPM) that can ensure curvature continuity. This LPM has already been developed in Chapter 4 and it is ready for use in this chapter. With the benchmark path planners developed, the next step is to develop continuous-curvature path-optimisation algorithms. This is the subject of Section 5.5. Finally, once the path-optimisation algorithms have been developed, they are then incorporated into the benchmark path planners to produce optimised path planners. This is done in Section 5.6. In this same section, the optimised path planners are analysed and conclusions regarding the developed optimised path-planning algorithms are drawn.

Prior to beginning the development and analyses of the optimised global path-planning algorithms according to the above outline, it is important to formalise the problem of optimised path planning addressed by this thesis and to introduce the notation and key definitions that will be used throughout the development of the algorithms. This is the subject of the next section.

5.3 Algorithm Development: Notation and Problem Formulation

In the introductory part of this chapter when the purpose of the optimised path planner was described, mention was made to a feasible path. This is a path connecting the initial configuration, \mathbf{q}_I , to the goal configuration, \mathbf{q}_G , without necessarily satisfying any optimality criteria. It was further mentioned that an optimised global path planner does not only find a path that is feasible, but one with the least cost, with the cost of a path measured by its length in this project. In this section we formalise the problems of feasible path planning and optimised path planning, as well as key definitions and notation used to represent paths computed by path-planning algorithms based on the RRT. The formalisation is necessary for the discussions on the algorithms to be developed in the next sections. We draw inspiration from definitions found in the work of Frazzoli and Karaman [36] for this formalisation.

Definition 1 (Global path): A global path from the initial configuration, $\mathbf{q}_I = [x_I \ y_I \ \theta_I]^T$, to the goal configuration, $\mathbf{q}_G = [x_G \ y_G \ \theta_G]^T$, is a continuous mapping $\tau : [0, 1] \mapsto \mathcal{C}_{\text{free}}$ where $\tau(0) = \mathbf{q}_I$, $\tau(1) = \mathbf{q}_G$ and is continuous everywhere between $\tau(0)$ and $\tau(1)$.

Definition 2 (Graph path): A graph path, also known as a *node path* and denoted by N , is a sequence of n graph vertices $\{v_0, v_1, v_2, \dots, v_{n-1}\}$ such that $(v_0, v_1), (v_1, v_2), \dots, (v_{n-2}, v_{n-1})$ are graph edges and each v_i is a unique graph vertex. In the context of sampling-based global path planning, the planner searches the configuration space \mathcal{C} by growing a graph through the sampling of configurations either deterministically or randomly. Each sampled configuration is added as a graph vertex if it is reachable from an existing node on the graph and at the same time the path (respecting the motion constraints of the robot) between these two configurations is added as an edge on the graph. Once the graph is grown to reach the goal configuration, the planner searches the graph for the *graph path* through which the goal has been reached from the root. The resulting graph path is a sequence of n configurations $\{\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_{n-1}\}$, where $\mathbf{q}_0 = \mathbf{q}_I$ and $\mathbf{q}_{n-1} = \mathbf{q}_G$.

In the remaining part of this section, we make use of the above definitions to formalise the problems of feasible path planning and optimised path planning.

Problem 1 (Feasible global path-planning problem): Given the inputs to the global path planning problem, namely the initial configuration, $\mathbf{q}_I = [x_I \ y_I \ \theta_I]^T$, the goal configuration, $\mathbf{q}_G = [x_G \ y_G \ \theta_G]^T$, and a map of the environment, m , find a path $\tau : [0, 1] \mapsto \mathcal{C}_{\text{free}}$ such that $\tau(0) = \mathbf{q}_I$ and $\tau(1) = \mathbf{q}_G$, if one exists; otherwise, report failure.

Definition 3 (Cost function of a path): If we let \mathcal{T} denote the set of all paths between any two configurations in \mathcal{C} of which a subset, $\mathcal{T}_{\text{free}}$, denotes the set of all feasible paths which are collision-free. Any path, τ , traversing the configurations $\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_{n-1}$ in \mathcal{C} is a concatenation of the paths, $\tau_1, \tau_2, \dots, \tau_n$, between the consecutive configuration pairs $(\mathbf{q}_0, \mathbf{q}_1), (\mathbf{q}_1, \mathbf{q}_2), \dots, (\mathbf{q}_{n-2}, \mathbf{q}_{n-1})$. The cost function c assigns a strictly positive cost to each path between two successive configurations, resulting in each of these paths having an associated cost c_{τ_i} . Since the cost function is strictly positive, it is also monotonic, i.e. $\forall \tau_a, \tau_b \in \mathcal{T}, c_{\tau_a} \leq c_{\tau_a|\tau_b}$, where $\tau_a|\tau_b$ denotes a concatenation of paths τ_a and τ_b . In this project the cost of a path is measured by its length.

Problem 2 (Optimised global path-planning problem). Given the inputs to the global path-planning problem, namely, the initial configuration, $\mathbf{q}_I = [x_I \ y_I \ \theta_I]^T$, the goal configuration, $\mathbf{q}_G = [x_G \ y_G \ \theta_G]^T$, a map of the environment, m , and a cost function, $c : \mathcal{T} \mapsto \mathbb{R}_{\geq 0}$, find a feasible and collision-free path, $\tau^* \in \mathcal{T}_{\text{free}}$, satisfying the condition $c_{\tau^*} = \min_{\tau \in \mathcal{T}_{\text{free}}} c_{\tau}$, if one exists; otherwise, report failure.

With the problems of feasible global path planning and optimised global path planning, together with important definitions and notation introduced, we now proceed to the development of the path-planning and path-optimisation algorithms that will constitute the optimised global path planners.

5.4 Algorithm Development: Adaptation of the Anytime RRT and the Informed RRT* for Continuous-curvature Path Planning

We now begin the development of the optimised global path-planning algorithms according to the outline given at the end of Section 5.2. As a first step in the development of the optimised global path planners to be developed in this thesis, this section focuses on the development of the anytime RRT and the informed RRT*, both adapted for curvature continuity. Again, the requirement for the planned paths to have a continuous-curvature profile emanates from the motion constraints of the vehicles targeted by this thesis – i.e. constrained angular velocity and angular acceleration.

The process employed in developing the adapted anytime RRT and the informed RRT* is to begin by developing the basic RRT algorithm, adapted for curvature continuity, and then building up on the foundation laid by this adapted basic RRT to come up with the adapted anytime and informed RRT* algorithms. Subsection 5.4.1 develops the continuous-curvature basic RRT, the continuous-curvature anytime RRT is developed in Subsection 5.4.2 and finally, the development of the continuous-curvature informed RRT* is the subject of Subsection 5.4.3.

5.4.1 Primitive Procedures and the Continuous-curvature Basic RRT

This subsection serves to lay a foundation for the two algorithms to be developed in the rest of this section, namely the anytime RRT and the informed RRT*. It does so by introducing the basic RRT, adapted for curvature continuity, and its primitive procedures.

We now introduce the procedures of the basic RRT, with the algorithm's pseudocode given in Algorithm 20.

Sampling: This procedure places a probability distribution over the configuration space, \mathcal{C} . Then at each iteration, it is used through the function call, `randomSample()` (line 7 of Algorithm 20) to generate a new random configuration, $\mathbf{q}_{\text{sampled}} \in \mathcal{C}$, towards which an attempt is then made to extend the tree.

Nearest neighbour selection: This is a function that takes the tree, $\mathcal{T} = (V, E)$ (in which each $v \in V$ is a vertex of the tree and each $e \in E$ is an edge of the tree), and a configuration, $q \in \mathcal{C}$, which is not part of the tree and returns the vertex, $\mathbf{q}_{\text{nearest}} \in V$, that is closest to q according to a given distance metric such as the Euclidean distance. In Algorithm 20, this function is utilised through the function call `nearestNeighbour($\mathcal{T}, \mathbf{q}_{\text{sampled}}$)` in line 8. Through this function call,

$\mathbf{q}_{\text{nearest}}$ is then earmarked as the vertex of the tree from which the extension towards $\mathbf{q}_{\text{sampled}}$ is to be attempted.

Steering: This function takes two configurations $\mathbf{q}_1, \mathbf{q}_2 \in \mathcal{C}$ and returns a new configuration, $\mathbf{q}_2' \in \mathcal{C}$, which is closer to \mathbf{q}_2 than \mathbf{q}_1 is. In the basic RRT, it is used to generate a configuration that is as close as possible to \mathbf{q}_2 while at the same time being at most a step size, η , from \mathbf{q}_1 . Thus the function minimises $\|\mathbf{q}_2' - \mathbf{q}_2\|$ while at the same time keeping \mathbf{q}_2' within a ball, $\mathcal{B} \subset \mathcal{C}$, of radius η centred at \mathbf{q}_1 . It is used through the function call `steer($\mathbf{q}_{\text{nearest}}, \mathbf{q}_{\text{sampled}}$)` in line 9 of Algorithm 20 to determine the location of a candidate vertex, $\mathbf{q}_{\text{candidate}}$, for which an attempt to add on the tree is to be made.

Local plan: Given two configurations $\mathbf{q}_1, \mathbf{q}_2 \in \mathcal{C}$, the function `LPM($\mathbf{q}_1, \mathbf{q}_2$)` returns a continuous-curvature local path from \mathbf{q}_1 to \mathbf{q}_2 . This path is generated using the continuous-curvature local path-planning algorithms implemented in Chapter 4. It is used to attempt a connection between $\mathbf{q}_{\text{nearest}}$ and $\mathbf{q}_{\text{candidate}}$ in line 10 of the algorithm.

Collision detection: Given a local path between two configurations, $\mathbf{q}_1, \mathbf{q}_2 \in \mathcal{C}$, computed through the algorithms implemented in Chapter 4, the Boolean function `collisionFree(LPM($\mathbf{q}_1, \mathbf{q}_2$))` returns `true` if the entirety of this path lies in $\mathcal{C}_{\text{free}}$, and `false` otherwise. It is called in line 10 of the algorithm to test the path from $\mathbf{q}_{\text{nearest}}$ to $\mathbf{q}_{\text{candidate}}$ for collisions.

Algorithm 20: `ccBasicRRT($\mathbf{q}_I, \mathbf{q}_G$)`

```

1 tree  $\mathcal{T}$ ;
2 Function initialiseRRT()
3    $V \leftarrow \{\mathbf{q}_I\}; E \leftarrow \emptyset$ ; // Add initial configuration to tree vertices.
4    $\mathcal{T} = (V, E)$ ;
5 Function extendRRT()
6   for iteration  $\leftarrow 1$  to maxIterations do
7      $\mathbf{q}_{\text{sampled}} \leftarrow \text{randomSample}()$ ; // Sample a random configuration.
8      $\mathbf{q}_{\text{nearest}} \leftarrow \text{nearestNeighbour}(\mathcal{T}, \mathbf{q}_{\text{sampled}})$ ; // Find nearest existing node.
9      $\mathbf{q}_{\text{candidate}} \leftarrow \text{steer}(\mathbf{q}_{\text{nearest}}, \mathbf{q}_{\text{sampled}})$ ; /* Configuration to be considered for addition to
      tree. */
10    if collisionFree(LPM( $\mathbf{q}_{\text{nearest}}, \mathbf{q}_{\text{candidate}}$ )) then
11       $\mathbf{q}_{\text{new}} \leftarrow \mathbf{q}_{\text{candidate}}$ ; // Candidate configuration confirmed as new.
12       $V \leftarrow V \cup \mathbf{q}_{\text{new}}$ ; // Add new configuration to vertex set.
13       $E \leftarrow E \cup \text{LPM}(\mathbf{q}_{\text{nearest}}, \mathbf{q}_{\text{new}})$ ; // Add local path to edge set.
14    if inGoalRegion( $\mathbf{q}_{\text{new}}$ ) then
15      return  $\mathcal{T}$ ; // Once tree reaches goal, solution is found.
16  return failure;
17 Function main()
18   initialiseRRT();
19   extendRRT();

```

As shown in Algorithm 20, the algorithm incrementally builds a tree of feasible paths by which configurations in \mathcal{C} can be reached. It begins by inserting the robot's initial configuration, \mathbf{q}_I , to the tree as the only reachable configuration through a call to the `initialiseRRT()` function in the `main()` function (line 18). Since the robot does not need to move any further to get to this configuration, no path is added to the edge set of the tree, thus it remains an empty set (line 3). Once the tree is initialised, the rest of the planning time is spent attempting to extend the tree so that it spreads through \mathcal{C} in search for the goal configuration. This is done through a call to the `extendRRT()` function (line 19). This function works by iteratively sampling a random configuration in \mathcal{C} (line 7) using the sampling procedure and then searching the tree using

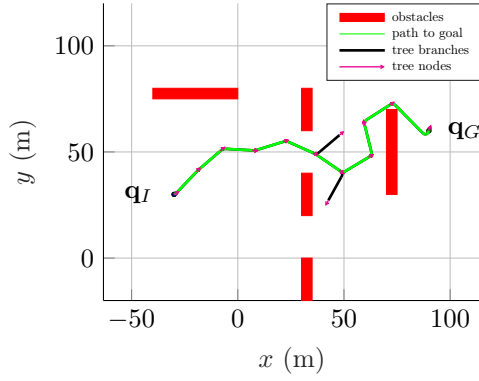
the nearest neighbour selection procedure for an existing vertex from which an extension can be attempted (line 8). A call to the steering method (line 9) determines the location of a candidate vertex to be considered for addition to the tree, after which a call to the local path planner computes the local path by which this candidate configuration can be reached from the nearest vertex on the tree. This path is then checked for collisions (line 10) and if it passes the test, the candidate configuration is added as the newest vertex to the set of tree vertices, V . The corresponding continuous-curvature local path is also added to the set of tree edges, E . The tree growth is continued until a new configuration that is within some user-specified goal tolerance is added to the tree, at which point a path from the initial configuration to the goal is obtained and the tree is returned (line 15). In the outlined formulation, the algorithm terminates immediately after such a vertex is added; however, if there is need to search for more paths to the goal, the algorithm can be reformulated so that the search can continue beyond this point, until the allocated number of iterations runs out.

Experimental results of independent runs of our MATLAB implementation of the developed algorithm are shown in the sub-figures of Figure 5.3 for an example global path-planning query in an environment representative of an office. At this stage, the purpose of the shown results is to show how the tree is grown and the incorporation of the continuous-curvature local paths to the basic RRT algorithm.

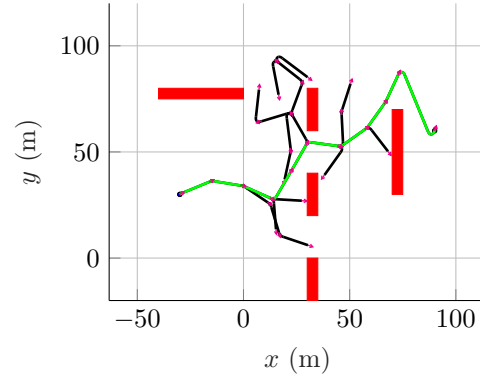
These example runs of the algorithm provide a demonstration of successful adaptation of the basic RRT algorithm for the generation of continuous-curvature paths, which can be executed by vehicles with a constrained angular velocity and angular acceleration. Moreover, a number of properties of the basic RRT algorithm are verified. These properties include the ability of the algorithm to find paths of different homotopic classes – a crucial attribute in ensuring avoidance of the planner getting trapped in a local minima. The term *homotopic* is taken from topology theory, where two continuous functions from one topological space are referred to as being from the same homotopic class if one of the functions can be continuously deformed into the other without violating constraints at any point of the deformation process. As examples, the paths in Figures 5.3(a) and 5.3(b) belong to the same homotopic class and the same applies to the paths in Figures 5.3(e) and 5.3(f). The ability of the algorithm to find a path to the goal quickly is also verified by the examples in Figures 5.3(a), 5.3(c) and 5.3(e) in which the planner finds a path to the goal within 15, 20 and 27 iterations respectively. Figures 5.3(b), 5.3(d) and 5.3(f) depict instances in which it took the planner significantly longer, compared to the other three cases to find the path to the goal. The significant difference in running time between the separate instances of algorithm demonstrates the randomness of the algorithm's runtime as a result of its dependence on random sampling. Another notable attribute of the algorithm is its disregard of the path cost both in terms of path length and proximity to obstacles when growing the search tree. The disregard of proximity to obstacles is due to the manner in which collision detection is handled. Without incorporating cost awareness into the collision detector, the search tree grows arbitrarily close to obstacles. Disregard for path cost in terms of path length results from the manner in which the existing node on the tree through which to connect the new node is selected. Without cost considerations imposed on the nearest neighbour selection function, each new node is added via the closest existing node in the tree, regardless of the cost of that node. The basic RRT algorithm solves the feasible path-planning problem without any notion for optimality, and as such does not take costs into consideration. The remaining two algorithms to be developed next in this section, namely the anytime RRT and the informed RRT* have a notion of optimality and as such they do take costs into consideration when building the search tree.

We have now developed the continuous-curvature basic RRT and discussed its primitive procedures. Just like in the development of the local path planner, the algorithms developed in this chapter are also implemented in MATLAB and C++, with MATLAB used for fast prototyping and testing while the C++ implementation will be used when the algorithms are deployed in ROS as part of the integrated autonomous navigation system in Part III. The working of the developed continuous-curvature basic RRT has been demonstrated through experimental results obtained from running our MATLAB implementation of the developed algorithm. These results also helped us verify a number of properties of the algorithm.

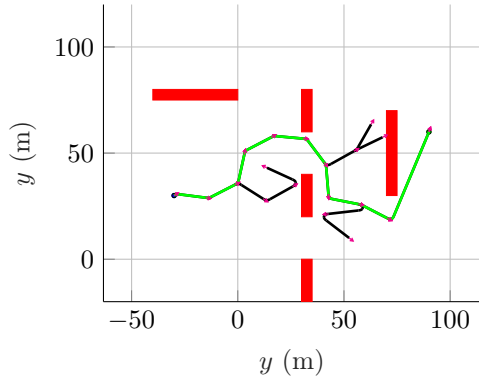
As stated in the beginning of this subsection, the main reason the continuous-curvature basic RRT has been discussed and developed is so that it can be used as a foundation for the development



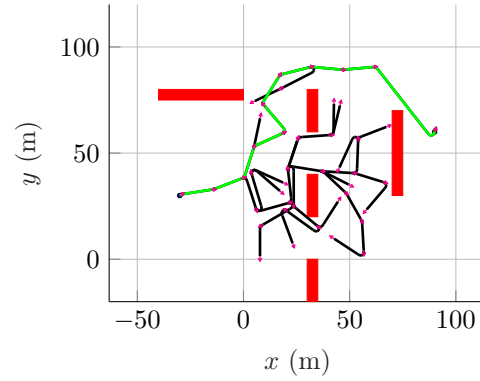
(a) Path found after 15 iterations with a path length of 163.6 m.



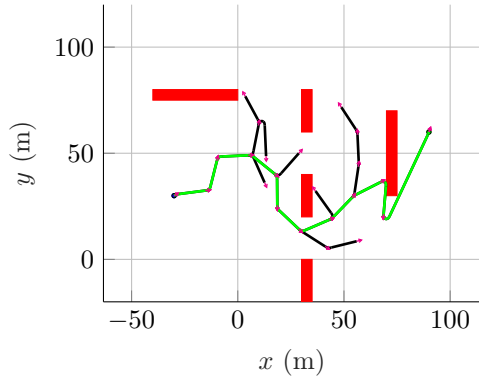
(b) Path found after 73 iterations with a path length of 175.5 m.



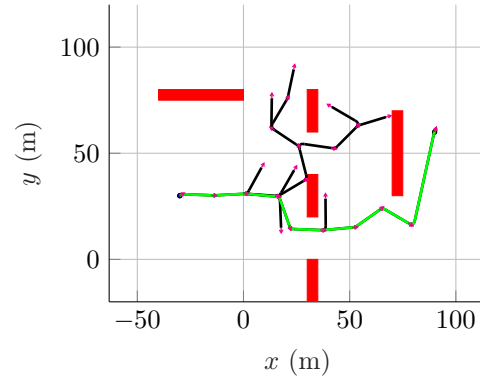
(c) Path found after 20 iterations with a path length of 184.6 m.



(d) Path found after 106 iterations with a path length of 184.8 m.



(e) Path found after 27 iterations with a path length of 205.2 m.



(f) Path found after 66 iterations with a path length of 169.4 m.

 Figure 5.3: Six independent instances of the CC basic RRT global path-planning algorithm for a planning query with $\mathbf{q}_I = [-30 \ 30 \ \frac{\pi}{9}]^T$ and $\mathbf{q}_G = [90 \ 60 \ \frac{\pi}{2}]^T$.

of the continuous-curvature anytime RRT and informed RRT* algorithms, which are respectively developed in Sections 5.4.2 and 5.4.3. We now proceed to the development of continuous-curvature anytime RRT.

5.4.2 Continuous-curvature Anytime RRT

We now adapt the anytime path-planning algorithm introduced by Ferguson and Stentz [18] for continuous-curvature path planning and point out places where it will be modified for our purpose.

The anytime RRT path planner works exactly the same way as the basic RRT until the first path to the goal is found. This is to ensure that a feasible path to the goal is found within minimal possible time, since the basic RRT is very efficient at finding a path. Thereafter, the remaining planning time is used to improve the existing solution by using the cost of the current best solution, c_{best} , to focus the sampling of new configurations so as to only consider configurations that can possibly lead to an improved solution compared to the current one. Pseudocode for our adapted version of the algorithm is given in Algorithm 21.

Algorithm 21: ccAnytimeRRT($\mathbf{q}_I, \mathbf{q}_G$)

```

1   $\mathcal{T} \leftarrow \emptyset$ ;  $c_b \leftarrow 0$ ;  $d_b \leftarrow 1$ ;  $c_{\text{best}} \leftarrow \infty$ ;  $\mathbf{Q}_{\text{soln}} \leftarrow \emptyset$ ; // Initialisation of variables.
2  Function growAnytimeRRT( $\mathcal{T}, c_{\text{best}}$ )
3      for iteration  $\leftarrow 1$  to maxIterationsPerTree do
4           $\mathbf{q}_{\text{sampled}} \leftarrow \text{informedSample}(\mathbf{q}_I, \mathbf{q}_G, c_{\text{best}})$ ; // Sample from informed subset.
5           $\mathbf{Q}_{\text{near}} \leftarrow \text{kNearestNeighbours}(\mathcal{T}, k, \mathbf{q}_{\text{sampled}})$ ; // k-nearest neighbours to sample.
6           $\mathbf{q}_{\text{new}} \leftarrow \text{null}$ ; // Initialisation of new configuration.
7          while  $\mathbf{Q}_{\text{near}} \neq \emptyset$  do // Attempt extension from cheapest among neighbours.
8               $\mathbf{q}_{\text{neighbour}} \leftarrow \text{argmin}_{\mathbf{q} \in \mathbf{Q}_{\text{near}}} \text{selCost}(\mathbf{q}, \mathcal{T}, \mathbf{q}_{\text{sampled}})$ ; // Cheapest among neighbours.
9               $\mathbf{Q}_{\text{near}} \leftarrow \mathbf{Q}_{\text{near}} \setminus \mathbf{q}_{\text{neighbour}}$ ; // Pop-out cheapest neighbour.
10              $\mathbf{Q}_{\text{ext}} \leftarrow \text{generateExtensions}(\mathbf{q}_{\text{neighbour}}, \mathbf{q}_{\text{sampled}})$ ; /* Generate a number of
                possible collision-free extensions using LPM and collision detector. */
11              $\mathbf{q}_{\text{candidate}} \leftarrow \text{argmin}_{\mathbf{q} \in \mathbf{Q}_{\text{ext}}} \text{cost}(\mathbf{q}_{\text{neighbour}}, \mathbf{q})$ ; // Choose cheapest extension
12             if  $\text{cost}(\mathbf{q}_I, \mathbf{q}_{\text{candidate}}) + g(\mathbf{q}_{\text{candidate}}, \mathbf{q}_G) < c_{\text{best}}$  then
13                  $\mathbf{q}_{\text{new}} \leftarrow \mathbf{q}_{\text{candidate}}$ ;
14                 break; // Exit if extension is within sub-optimality bound.
15             if  $\mathbf{q}_{\text{new}} \neq \text{null}$  then
16                  $V \leftarrow V \cup \mathbf{q}_{\text{new}}$ ; // Add new node to vertex set.
17                  $E \leftarrow E \cup \text{LPM}(\mathbf{q}_{\text{neighbour}}, \mathbf{q}_{\text{new}})$ ; // Add local path to edge set.
18                 if  $\text{inGoalRegion}(\mathbf{q}_{\text{new}})$  then // New node in vicinity of goal configuration.
19                      $\text{localPath} \leftarrow \text{LPM}(\mathbf{q}_{\text{new}}, \mathbf{q}_G)$ ; // Local path computation.
20                      $c_{\text{soln}} \leftarrow \text{cost}(\mathbf{q}_I, \mathbf{q}_{\text{new}}) + \text{cost}(\mathbf{q}_{\text{new}}, \mathbf{q}_G)$ ; // Cost of new solution.
21                     if  $c_{\text{soln}} < c_{\text{best}}$  then
22                         if  $\text{collisionFree}(\text{localPath})$  then
23                              $V \leftarrow V \cup \mathbf{q}_G$ ; // Add goal configuration to vertex set.
24                              $E \leftarrow E \cup \text{localPath}$ ; // Add local path to edge set.
25                              $\mathbf{Q}_{\text{soln}} \leftarrow \mathbf{Q}_{\text{soln}} \cup \text{getPath}(\mathbf{q}_I, \mathbf{q}_G)$ ; // Store new solution.
26                             return  $c_{\text{soln}}$ 
27             return  $\text{null}$ ; // Iterations exhausted without finding a cheaper solution.
28 Function main()
29     repeat
30          $\text{reInitialiseRRT}(\mathcal{T})$ ; // Clear tree and update root.
31          $c_{\text{new}} \leftarrow \text{growAnytimeRRT}(\mathcal{T}, c_{\text{best}})$ ;
32         if  $c_{\text{new}} \neq \text{null}$  then
33              $\text{postCurrentSolution}(\mathbf{Q}_{\text{soln}})$ ; // Publish the current best solution.
34              $c_{\text{best}} \leftarrow c_{\text{new}}$ ; // Update cost of current best solution.
35              $c_b \leftarrow c_b + \delta_c$ ; // Update cost bias (increase it gradually as tree grows).
36             if  $c_b > 1$  then
37                  $c_b \leftarrow 1$ ; // Prevent cost bias from getting greater than 1.
38              $d_b \leftarrow d_b - \delta_d$ ; // Update distance bias (increase it gradually).
39             if  $d_b < 1$  then
40                  $d_b \leftarrow 0$ ; // Prevent distance bias from being negative.
41     until  $\text{planningTimeElapsed}()$ ;

```

The idea of the anytime RRT is to build a series of independent RRT search trees, where each new tree strives to find a solution that is better than all solutions found by previous trees

– the cost of the solution found by each tree is used to enforce a sub-optimality bound for the subsequent search. The function `reInitialiseRRT`, called within the main function in line 30, essentially clears the tree structure grown by a previous tree and updates the root of the tree to the current position of the robot or another position in the configuration space from which planning is desired to originate. The algorithm encourages each successive tree to find a better path by altering the manner in which the tree is grown through the `growAnytimeRRT` function defined in lines 2–26 and called at line 31 in the main function. The primary way in which the tree growth is altered is by guiding the manner in which sample configurations are generated. In the original formulation of the anytime RRT, this guidance was done through the use of a heuristic function and rejection sampling. The heuristic function used is similar to that used by the A* algorithm discussed in Subsection 2.5.1.2.2. It approximates the cost from the root of the tree, \mathbf{q}_I , to the goal configuration, \mathbf{q}_G , via the sampled configuration, $\mathbf{q}_{\text{sampled}}$. It is of the form:

$$f(\mathbf{q}_{\text{sampled}}) = h(\mathbf{q}_I, \mathbf{q}_{\text{sampled}}) + g(\mathbf{q}_{\text{sampled}}, \mathbf{q}_G), \quad (5.2)$$

where $h(\mathbf{q}_I, \mathbf{q}_{\text{sampled}})$, also known as the heuristic cost-to-come, represents an estimate of the cost of the path from the root of the tree to the sampled configuration and $g(\mathbf{q}_{\text{sampled}}, \mathbf{q}_G)$, also known as the heuristic cost-to-go, estimates the cost of the path from the sampled configuration to the goal configuration. The sampling guidance worked by sampling configurations uniformly in the configuration space, testing the heuristic cost of each sampled configuration against the current best cost, c_{best} , and only considering the sample for tree extension if $f(\mathbf{q}_{\text{sampled}}) < c_{\text{best}}$; otherwise rejecting it. This is repeated until an acceptable sample is found or the maximum number of samples is reached. While overall this rejection sampling approach contributes towards the improvement of future solutions, time is wasted in sampling and testing each of the rejected samples. In our adapted version of the anytime RRT algorithm, we use one aspect of the informed RRT* to eliminate this time wastage by directly sampling the admissible heuristic region which turns out to be an ellipsoid for problems in which the cost is the path length and the heuristic cost-to-come and cost-to-go are approximated by straight-line distances [20]. In Algorithm 21, this sampling guidance is implemented in the `informedSample` function called in line 4.

The second method by which the tree growth is changed in the anytime RRT is by altering the vertex selection method when adding a new configuration to the tree. Instead of simply selecting the closest node like in the basic RRT, the vertex selection method is altered so as to favour nodes with both least cumulative cost from the root of the search tree and shorter distance to the sample according to a node selection function of the form:

$$\text{selCost}(\mathbf{q}) = c_b * \text{cost}(\mathbf{q}_I, \mathbf{q}) + d_b * \text{distance}(\mathbf{q}, \mathbf{q}_{\text{sampled}}), \quad (5.3)$$

where c_b and d_b are known as bias factors, with c_b being a *cost-bias factor* and d_b a *distance-bias factor* – both bias factors are constrained to be in the range (0,1). This cost is computed for each of the k nearest neighbours to $\mathbf{q}_{\text{sampled}}$ and the neighbour with the lowest cost is selected for extension. These bias parameters are initially set to $d_b = 1$ and $c_b = 0$, thus resulting in a pure nearest-neighbour selection as in the basic RRT and hence retaining the RRT's probabilistic completeness property. Each time a better path to the goal is found, the bias parameters are altered by increasing c_b by some value, δ_c (line 35) while at the same time reducing d_b by some value, δ_d (line 38). Again, the bias parameters are constrained from being negative and greater than 1 (lines 36–37, and lines 39–40). As a result, as more and more better paths to the goal are found, c_b approaches 1 and d_b approaches zero. This makes the cost of neighbouring nodes to become more important than their distance from the sampled configuration in their selection for tree extension. When c_b eventually becomes 1 and d_b becomes zero, the nearest neighbour selection procedure becomes purely based on the cost of each neighbour from the root, with no regard to its proximity to $\mathbf{q}_{\text{sampled}}$. The effect is that early on, costly solutions are produced (when c_b is smaller and d_b , larger) and then cheaper solutions are found later on (when c_b becomes larger and d_b , smaller). This effect of the bias parameters is in line with the principle of anytime path planning, where the desire is to find solutions quicker, though expensive, early on and then improve the solutions later with extra available planning time.

Lastly, tree growth is changed by altering the manner in which an extension from the selected neighbour towards the sampled configuration is done. The original anytime RRT presented two

approaches for extending the tree once an existing neighbouring node, $\mathbf{q}_{\text{neighbour}}$, has been selected for extension. Instead of simply attempting an extension from the neighbour in the general direction of $\mathbf{q}_{\text{sampled}}$ as in the basic RRT, the first approach starts by attempting this extension and only considers it if it could possibly lead to a solution with a lower cost than c_{best} , i.e.

$$\text{cost}(\mathbf{q}_I, \mathbf{q}_{\text{neighbour}}) + \text{cost}(\mathbf{q}_{\text{neighbour}}, \mathbf{q}_{\text{candidate}}) + g(\mathbf{q}_{\text{candidate}}, \mathbf{q}_G) < c_{\text{best}}, \quad (5.4)$$

where, $\mathbf{q}_{\text{candidate}}$, known as the candidate node, is the configuration at the end of the extension, $\text{cost}(\mathbf{q}_I, \mathbf{q}_{\text{neighbour}}) + \text{cost}(\mathbf{q}_{\text{neighbour}}, \mathbf{q}_{\text{new}})$ gives the cost-to-come for $\mathbf{q}_{\text{candidate}}$ with $\mathbf{q}_{\text{neighbour}}$ as its parent and $g(\mathbf{q}_{\text{candidate}}, \mathbf{q}_G)$ is the heuristic cost-to-go for $\mathbf{q}_{\text{candidate}}$. If this condition is not satisfied for the extension in the general direction of $\mathbf{q}_{\text{sampled}}$, then extensions that do not lead directly towards $\mathbf{q}_{\text{sampled}}$ are considered. These alternative extensions are attempted until an extension resulting in a candidate node, $\mathbf{q}_{\text{candidate}}$, satisfying Equation 5.4 is achieved or until the number of allowed extension attempts is exhausted. Unlike the first, the second extension approach generates a large initial set of possible extensions and then takes the cheapest among the generated extensions – again, the candidate node, $\mathbf{q}_{\text{candidate}}$, is checked to ensure that it satisfies Equation 5.4 before being added as a new node, \mathbf{q}_{new} , on the tree. The first approach is generally faster in generating an extension and the second one, though slow, produces less costly solutions than the first. In Algorithm 21, the extension process occurs in lines 10–14. The generation of possible collision-free extensions through calls to the LPM and the collision detector is accomplished by the `generateExtensions` function call (line 10). The node at the endpoint of the cheapest extension (line 11) is the new candidate node, pending verification that it satisfies the current sub-optimality bound given by Equation 5.4 which happens in line 12. With $\mathbf{q}_{\text{candidate}}$ passing this test, it is confirmed as the new node, \mathbf{q}_{new} , and it is added on the tree (lines 15–17); otherwise the extension process is started afresh with the next-cheapest neighbour among the k nearest neighbours, until a valid extension is found or until extensions have been attempted from all the k nearest neighbours. A successful tree extension is concluded by an attempt to add the goal configuration to the tree if the new node is within a user specified threshold to the goal (line 18). The attempt to add the goal configuration to the tree involves generating a local path from \mathbf{q}_{new} to \mathbf{q}_G (line 19) and checking if the cost of the path to the goal satisfies the sub-optimality bound (line 21). With sub-optimality bound satisfied and the generated local path collision-free, the goal configuration is added to the tree and a new, cheaper solution is found (lines 23–25). The cost of the new solution is returned by the `growAnytimeRRT` function. In the event that the number of allocated iterations for the growth of each anytime RRT tree elapses without a solution with a cheaper cost found, the `growAnytimeRRT` function returns a `null` value for the cost of the new solution.

The main function of the anytime RRT algorithm makes use of the variables associated with the above-discussed `growAnytimeRRT` procedure to facilitate the growth of the series of search trees until the planning time runs out. Each new path is published through the function call `postCurrentSolution()` (line 33) so that it can be available for execution, should planning time run out. The respective algorithm parameters are updated as discussed earlier with each newly-found path (lines 34–40).

The anytime RRT algorithm just described has two extremes with regard to the quality of the paths it generates. These extremes are determined by the values of the cost-bias factors and the nature of the tree extension process. As explained earlier in this subsection, with the cost-bias factor, c_b , zero and the distance-bias factor, d_b , 1, the nearest-neighbour selection is exactly the same as that of the basic RRT, simply selecting the nearest existing tree node, regardless of its cost from the root, for extension – this can be also achieved by simply setting $k = 1$, i.e. considering only one neighbour for extension. With $c_b = 1$ and $d_b = 0$, the nearest-neighbour selection is purely based on the cost of a node from the root; making nodes with cheaper cost-from-root preferred for extension and as a result, producing cheaper paths. With regard to the extension process, among the two approaches for tree extension presented in the original anytime RRT, the first, which starts by attempting an extension that leads in the general direction of $\mathbf{q}_{\text{sampled}}$, only attempting alternative extensions that do not lead in this direction if the direct extension is in collision, is faster. However, it produces paths that are expensive compared to the second one which generates a large number of initial extensions and picks the cheapest. It is interesting to consider an extension procedure that is exactly the same as that of the basic RRT, i.e. one that simply attempts the

extension in the general direction of $\mathbf{q}_{\text{sampled}}$, without considering alternative extensions. This extension procedure results in quicker extensions, but with paths that are expensive than both the extension approaches presented by the original anytime RRT. At this point, it is clear that by choosing the values of the bias factors as well as the nature of the extension process, the quality of paths returned by the anytime RRT and the time taken for planning can be varied. Particularly, at one extreme, the algorithm can be configured so that both the neighbour selection and the extension process is exactly the same as those of the basic RRT, resulting in quicker planning, though producing costly solutions. This behaviour can be achieved by setting $k = 1$ or by keeping the bias factors fixed at $c_b = 0$ and $d_b = 1$ throughout the entire planning period. At the other extreme, with the neighbour selection being purely cost-based and the extension process preferring cheaper extensions, planning is slow, but produces cheaper solutions. This corresponds to $k > 1$ and the bias factors fixed at $c_b = 1$ and $d_b = 0$ throughout the entire planning period. In-between these extremes, with $k > 1$, planning time and the quality of generated paths increases as c_b and d_b approach 1 and 0 respectively. The ability to vary the quality of generated paths in the manner described renders the anytime RRT useful for generating a rich set of paths with varying optimality. These paths will be used as initialisations to the path-optimisation algorithms to be developed in Section 5.5, providing a rich set of initialisations through which the path optimisers can be tested and verified. The ability to vary the quality of generated paths will also serve as an avenue through which the effect of incorporating the path optimisation step in the anytime RRT will be tested.

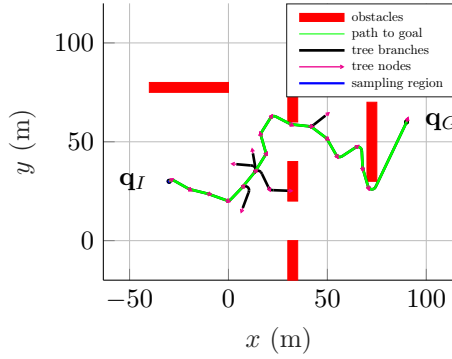
Experimental results showing an example run of the anytime algorithm adapted to the continuous-curvature case with informed sampling are shown in Figure 5.4. To demonstrate the effectiveness of the algorithm in converging towards the optimal path by iteratively bounding the sampling space, in the illustrated path planning scenario, the continuous-curvature anytime RRT was configured to behave exactly as the basic RRT both in neighbour selection and tree extension. Under these settings, the algorithm generates costly solutions as explained in the preceding discussion. By showing the effectiveness of the of the algorithm in converging towards the optimal solution even when using the cost of expensive paths to iteratively bound the sampling region, we illustrate the effectiveness of the principle behind it, without reliance on the quality of initial paths.

In this particular run, the algorithm finds the first path after 26 iterations (Figure 5.4(a)); the cost of this path is then used to restrict the sampling space for the second tree as depicted by the elliptic region in Figure 5.4(b) where the second tree is grown. The second tree finds a path with a lower cost than that found by the first RRT and thus the new path cost is used to further reduce the sampled region for the growth of the third RRT in Figure 5.4(c). This trend repeats in all other successive searches, with each successive search finding a path with a cheaper cost than the previous one and hence tending towards the optimal path.

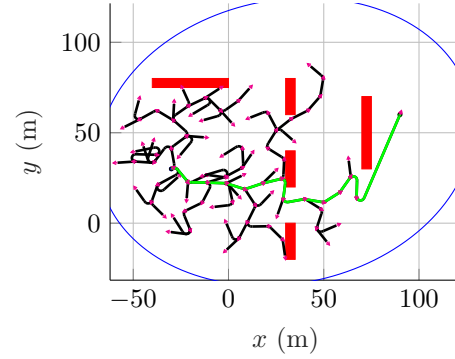
The presented results demonstrate successful adaptation of the anytime RRT path-planning algorithm for continuous-curvature path planning. The algorithm indeed has the ability to improve an initial RRT solution by iteratively reducing the sampled subset of the configuration space. The algorithm was demonstrated using parameter values that cause it to extend the tree exactly the same way as the basic RRT. The presented results show the effectiveness of the algorithm even under this setting. With parameter values that incorporate cost considerations when selecting the nearest neighbour and when extending the tree, the performance of the algorithm can only improve as the sampled ellipsoidal subset of the configuration space would be reduced more rapidly, encouraging the algorithm to find better solutions. Furthermore, if a path-optimisation step is applied to each found path before being used to bound region sampled in the subsequent search, the rate at which the ellipsoid reduces could also possibly improve. This application of path optimisation to each path found by the algorithm is the subject of Section 5.6. The next subsection presents the development the continuous-curvature informed RRT*.

5.4.3 Continuous-curvature Informed RRT*

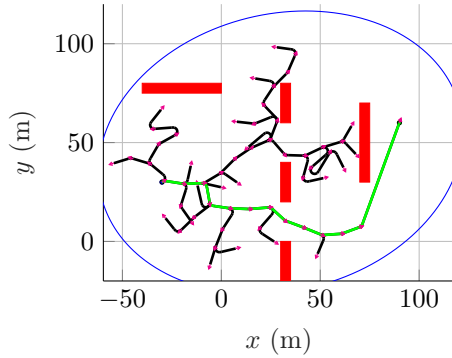
In this subsection we move on to develop the second benchmark path-planning algorithm pointed out in the outline given at the end of Section 5.2, the informed RRT* adapted for curvature continuity. This adaptation is again done by making use of the continuous-curvature LPM developed in Chapter 4 to adapt the original informed RRT*, with the goal of generating paths that can



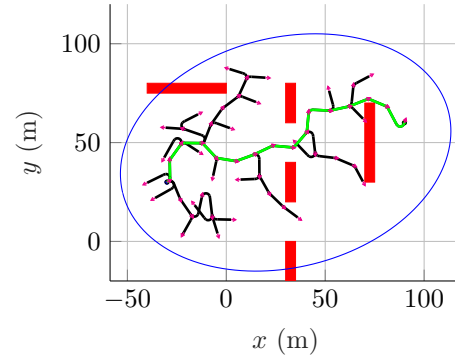
(a) Path found after 26 iterations with a path length of 199.4 m.



(b) Path found after 146 iterations with a path length of 186.8 m.



(c) Path found after 218 iterations with a path length of 169.7 m.

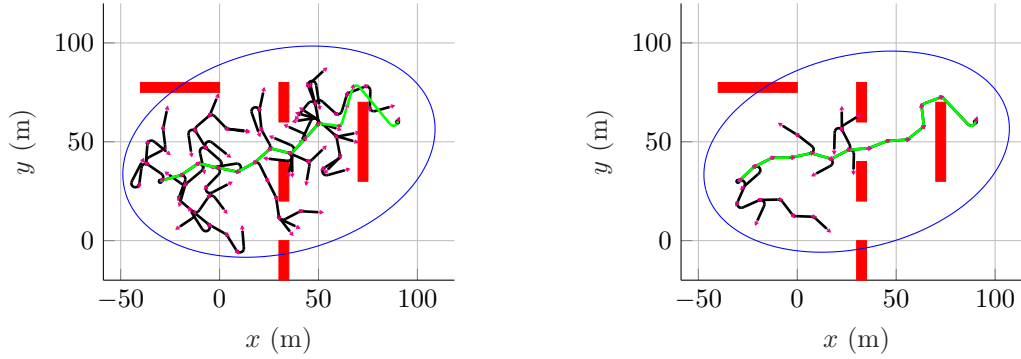


(d) Path found after 291 iterations with a path length of 160.7 m.

Figure 5.4: An example run of the anytime RRT algorithm adapted for the continuous curvature case for a path planning query with $\mathbf{q}_I = [-30 \ 30 \ \frac{\pi}{9}]^T$ and $\mathbf{q}_G = [90 \ 60 \ \frac{\pi}{2}]^T$. Each run uses the cost of the solution found in the previous run to bound the sampling space and then begins a new search tree growth in this reduced search space. The figure continues on the next page.

be accurately followed by vehicles with a constrained angular velocity and angular acceleration in the absence of disturbances. The original informed RRT* made use of a straight-line LPM and therefore generates paths made up of a concatenation of straight lines. Such paths cannot be followed accurately by vehicles with constrained angular velocity and angular acceleration, even in the absence of disturbances.

The informed RRT* [20] is, in its own right, an important sampling-based path-planning algorithm that marked a historical change in the research direction of sampling-based path planning. It is based on two other similarly important sampling-based path-planning algorithms, which also marked historical changes in sampling-based path planning research. These are the anytime RRT [18] and the RRT* [36]. The anytime RRT introduced the idea of striving towards the optimal solution by performing a series of independent RRT searches, with the cost of the solution found by each search used to bound the region of the configuration space sampled by the subsequent search. As seen in the preceding subsection, this principle focuses the search, encouraging each subsequent search to find a better solution compared to the previous one. The anytime RRT introduced the notion of a *sub-optimality bound* for each subsequent search – which is that each subsequent search is guaranteed to find a solution that has a lower cost than that found by the previous search. On the other hand, the RRT* introduced provable guarantees for finding the optimal path in sampling-based path planning. The RRT* is said to be asymptotically optimal. This means that it is guaranteed to find the optimal path as the number of samples tends to infinity. As mentioned in Subsection 2.6.4.1, an important algorithmic change introduced by the RRT* is the introduction of a *rewiring process* by which it continually improves the paths from the root of the tree to all nodes of the tree by rewiring a subset of the tree with each newly-added node. The reliance of the



(e) Path found after 479 iterations with a path length of 157.3 m.

(f) Path found after 520 iterations with a path length of 147.7 m.

Figure 5.4: Continued from previous page.

of the RRT* on the number of samples approaching infinity and its principle of striving towards optimality by continually improving paths from the root to all tree nodes through rewiring makes its rate of convergence towards the optimal solution slow. The informed RRT* is among successors of the RRT* which aim to accelerate its rate of convergence. Among these successors, the informed RRT* turned out to be superior by introducing a technique for accelerating the rate of convergence of the RRT* without violating any assumption on which the RRT* is based – as mentioned earlier in Subsection 2.6.4.2, another notable attempt at accelerating the RRT*'s rate of convergence, the RRT*-smart, did so by violating key RRT* assumption, i.e. uniform sampling.

To accelerate the rate of convergence of the RRT*, the informed RRT*, combines properties of the anytime RRT with those of the RRT*. In particular, the property inherited from the anytime RRT is the iterative reduction of the sampled space with each newly-found solution – only focusing on configurations that can possibly improve the current solution. Unlike the anytime RRT, the informed RRT* does not grow a series of independent trees, but rather maintains one tree throughout the planning process, allowing it to reuse search efforts from previous searches in subsequent ones. From the RRT*, the informed RRT* inherits the ability to strive towards finding the optimal solution through continually improving the paths from the root to all nodes in the search tree by rewiring a subset of the tree with each newly-added node. By combining these properties, the informed RRT* accelerates the convergence rate by sampling only a small, iteratively-reducing subset of the configuration space containing configurations that can possibly improve the current solution unlike the RRT*, which samples the entire configuration space throughout the planning period. With sampling of new configurations restricted to happen in the smaller, iteratively-reducing subset of the configuration space – known as the *informed subset*; tree extension and rewiring also occurs in this subset. This makes the informed RRT* quicker to converge towards the optimal solution than the RRT*.

The rest of this subsection presents our adaptation of the informed RRT* to continuous-curvature path planning. The pseudocode of the adapted algorithm is shown in Algorithm 22. Line 1 initialises the tree structure, a set to store found solution paths, \mathbf{Q}_{soln} , and the cost of the current best solution, c_{best} . With the tree initialised, the rest of the planning time is spent growing and rewiring it. In each iteration, the cost of the current best solution is used to bound the sampled subset of the configuration space. A sample configuration, $\mathbf{q}_{\text{sampled}}$, towards which an extension of the tree is to be attempted is then drawn from this informed subset (line 3). With the sample configuration drawn, the nearest existing tree node to it, $\mathbf{q}_{\text{nearest}}$ is determined (line 4). A call to the steering method (line 5) is then used to determine a new configuration, \mathbf{q}_{new} , for which an attempt to be added as a node to the tree is then made. The attempt to add \mathbf{q}_{new} as a tree vertex begins with a call to the LPM to compute a continuous-curvature local path from $\mathbf{q}_{\text{nearest}}$ to \mathbf{q}_{new} . The computed continuous-curvature local path is also tested for the absence of collisions before its is added to the tree (line 7). On successful addition of the new node to the vertex set of the tree (line 8), the informed RRT* checks if the initial continuous-curvature local path (i.e.

Algorithm 22: ccInformedRRT*($\mathbf{q}_I, \mathbf{q}_G$)

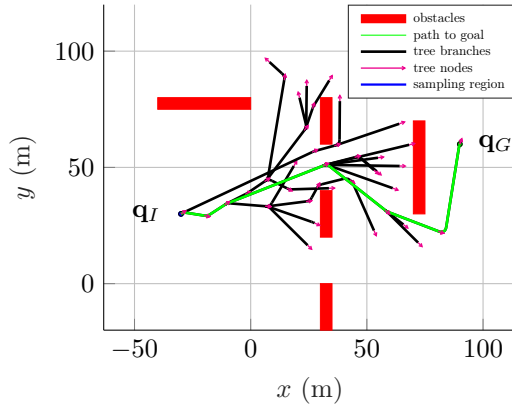
```

1   $V \leftarrow \{\mathbf{q}_I\}; E \leftarrow \emptyset; \mathcal{T} = (V, E); \mathbf{Q}_{\text{soln}} \leftarrow \emptyset; c_{\text{best}} \leftarrow \infty$  ; // Initialisations.
2  for iteration  $\leftarrow 1$  to maxIterations do
3       $\mathbf{q}_{\text{sampled}} \leftarrow \text{informedSample}(\mathbf{q}_I, \mathbf{q}_G, c_{\text{best}})$  ; // Sample from informed subset.
4       $\mathbf{q}_{\text{nearest}} \leftarrow \text{nearestNeighbour}(\mathcal{T}, \mathbf{q}_{\text{sampled}})$  ; // Closest existing node on tree.
5       $\mathbf{q}_{\text{new}} \leftarrow \text{steer}(\mathbf{q}_{\text{nearest}}, \mathbf{q}_{\text{sampled}})$  ; // New configuration to try extension to.
6       $\text{localPath} \leftarrow \text{LPM}(\mathbf{q}_{\text{nearest}}, \mathbf{q}_{\text{new}})$  ; // Local path computation.
7      if collisionFree(localPath) then
8           $V \leftarrow V \cup \mathbf{q}_{\text{new}}$  ; // Add new configuration to vertex set.
9           $E \leftarrow E \cup \text{localPath}$  ; // Add local path to edge set.
10          $\mathbf{Q}_{\text{near}} \leftarrow \text{near}(\mathcal{T}, \mathbf{q}_{\text{new}}, r_{\text{RRT}^*})$  ; // Nodes within ball radius to  $\mathbf{q}_{\text{new}}$ .
11          $c_{\text{min}} \leftarrow \text{cost}(\mathbf{q}_I, \mathbf{q}_{\text{nearest}}) + \text{cost}(\mathbf{q}_{\text{nearest}}, \mathbf{q}_{\text{new}})$  ; //  $\mathbf{q}_{\text{new}}$ 's cost via  $\mathbf{q}_{\text{nearest}}$ .
12         foreach  $\mathbf{q}_{\text{near}} \in \mathbf{Q}_{\text{near}}$  do // Try finding a cheaper parent for new node.
13              $\text{localPath} \leftarrow \text{LPM}(\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}})$  ; // Local path computation.
14              $c_{\text{new}} \leftarrow \text{cost}(\mathbf{q}_I, \mathbf{q}_{\text{near}}) + \text{cost}(\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}})$ ;
15             if  $c_{\text{new}} < c_{\text{min}}$  then // Tested parent actually cheaper.
16                 if collisionFree(localPath) then
17                      $E \leftarrow E \setminus \text{localPath}$  ; // Delete existing edge.
18                      $E \leftarrow E \cup \text{localPath}$  ; // Replace with cheaper edge.
19                      $c_{\text{min}} \leftarrow c_{\text{new}}$  ; // Update  $\mathbf{q}_{\text{new}}$ 's cost via current cheaper parent.
20         foreach  $\mathbf{q}_{\text{near}} \in \mathbf{Q}_{\text{near}}$  do // Rewire tree.
21              $c_{\text{near}} \leftarrow \text{cost}(\mathbf{q}_I, \mathbf{q}_{\text{near}})$  ; //  $\mathbf{q}_{\text{near}}$ 's cost via current parent.
22              $\text{localPath} \leftarrow \text{LPM}(\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{near}})$  ; // Local path computation.
23              $c_{\text{new}} \leftarrow \text{cost}(\mathbf{q}_I, \mathbf{q}_{\text{new}}) + \text{cost}(\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{near}})$  ; //  $\mathbf{q}_{\text{near}}$ 's cost via  $\mathbf{q}_{\text{new}}$ .
24             if  $c_{\text{new}} < c_{\text{near}}$  then
25                 if collisionFree(localPath) then
26                      $\mathbf{q}_{\text{parent}} \leftarrow \text{getParent}(\mathbf{q}_{\text{near}})$  ; // Existing parent of  $\mathbf{q}_{\text{near}}$ .
27                      $E \leftarrow E \setminus \text{getEdge}(\mathbf{q}_{\text{parent}}, \mathbf{q}_{\text{near}})$  ; // Delete existing edge.
28                      $E \leftarrow E \cup \text{localPath}$  ; // Replace with cheaper edge.
29         if inGoalRegion( $\mathbf{q}_{\text{new}}$ ) then // New node in vicinity of goal.
30              $\text{localPath} \leftarrow \text{LPM}(\mathbf{q}_{\text{new}}, \mathbf{q}_G)$  ; // Local path computation.
31             if collisionFree(localPath) then
32                  $V \leftarrow V \cup \mathbf{q}_G$  ; // Add goal configuration as a vertex on tree.
33                  $E \leftarrow E \cup \text{localPath}$  ; // Add local path as an edge on tree.
34                 if  $\text{cost}(\mathbf{q}_I, \mathbf{q}_G) < c_{\text{best}}$  then // New solution cheaper.
35                      $\mathbf{Q}_{\text{soln}} \leftarrow \mathbf{Q}_{\text{soln}} \cup \text{getPath}(\mathbf{q}_G)$  ; // Store new solution.
36                      $c_{\text{best}} \leftarrow \text{cost}(\mathbf{q}_I, \mathbf{q}_G)$  ; // Update cost of current best solution.
37 return  $\mathcal{T}$ 

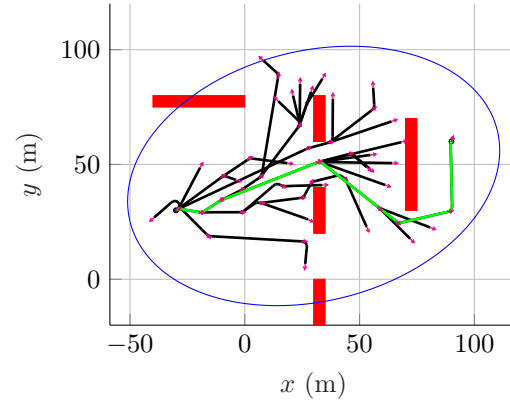
```

the one connecting \mathbf{q}_{new} via $\mathbf{q}_{\text{nearest}}$) can be replaced by a cheaper continuous-curvature local path by, instead, connecting \mathbf{q}_{new} via any other existing node that lies within a ball of radius r_{RRT^*} to \mathbf{q}_{new} . If a cheaper alternative parent for \mathbf{q}_{new} is found that results in a cheaper, collision-free path compared to the path from \mathbf{q}_{near} , then \mathbf{q}_{new} is added via this cheaper parent, otherwise, the connection via $\mathbf{q}_{\text{nearest}}$ is maintained. This is performed in lines 9–19. Once the new node and its edge are added to the tree, the new node, \mathbf{q}_{new} , is considered as a replacement for parent nodes of neighbouring nodes lying within a ball of radius r_{RRT^*} to it. A parent of a node in \mathbf{q}_{new} 's neighbourhood is replaced by \mathbf{q}_{new} if reaching the neighbouring node via \mathbf{q}_{new} is cheaper than reaching it via its current parent. This is the rewiring process and it occurs in lines 20–28 of the algorithm. Each iteration of the algorithm is then concluded by determining if the newly-added node is within the vicinity of the goal (line 30) and if so, an attempt to add the goal to the tree is made and if successful, a new solution is found and the \mathbf{q}_G is added to the tree (lines 32–33). The

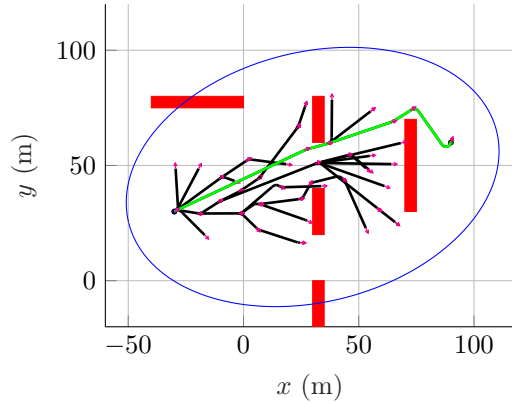
new solution is added to the solution set, \mathbf{Q}_{soln} , and its cost is used to update the current best cost, c_{best} , if it is cheaper than the current best cost (lines 34–36). The updated value of c_{best} is used further reduce the search space in the subsequent iteration through informed configuration space sampling.



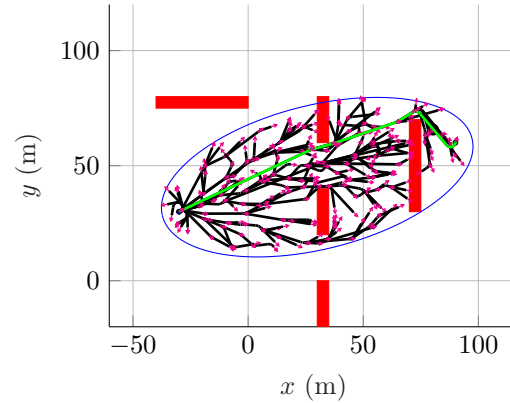
(a) Path found after 47 iterations with a path length of 164.8 m.



(b) Path found after 61 iterations with a path length of 164.5 m.



(c) Path found after 64 iterations with a path length of 138.6 m.



(d) Path found after 370 iterations with a path length of 137.1 m.

Figure 5.5: An example run of the informed RRT* algorithm adapted for the continuous curvature case for a path-planning query with $\mathbf{q}_I = [-30 \ 30 \ \frac{\pi}{9}]^T$ and $\mathbf{q}_G = [90 \ 60 \ \frac{\pi}{2}]^T$. The tree is maintained throughout the planning period. Each found solution is used to bound the sampling space in subsequent runs as depicted by the blue ellipse. Once the sampling region is bounded, nodes that fall outside the bounded space are pruned from the tree as they cannot possibly improve the current solution.

Experimental results showing an example run of our MATLAB implementation of the developed algorithm are given in Figure 5.5, in a similar way as those presented for the adapted anytime path planner. Just like in the results presented to demonstrate the continuous-curvature anytime path planner in the previous subsection, at this stage, the purpose of the depicted results is to demonstrate the working of the adapted informed RRT* algorithm. In the presented global path-planning scenario, the algorithm finds an initial path to the goal in 47 iterations as depicted in Figure 5.5(a). Once the initial path is found, the sampling space is bounded using the cost of this solution in the subsequent search as shown by the elliptical region in Figure 5.5(b). Through further extension and rewiring of the tree in the second search, after 61 iterations, the algorithm succeeds in finding a better path to the goal than the existing one. As such, the cost of the new path is used to further restrict the search space in the third search as demonstrated by the slight change in the size of the bounded elliptical region in Figure 5.5(c). The third search succeeds in

finding a path with a significantly lower cost than the previous one. This results in a dramatic reduction of the sampling region for the fourth search as seen in Figure 5.5(d). An important capability demonstrated in the result of the third search is the ability of the algorithm to find a path of a different homotopy class during tree expansion and rewiring as it improves the existing solution. Figure 5.5(d) shows the fourth and last search, which finds a solution path that is, by inspection, much closer to optimal at the 370th iteration.

The planning example shown demonstrates successful implementation of the informed RRT* adapted for continuous-curvature global path planning. Its ability to improve the solution path by both reducing the sampling space with each found solution as well as by rewiring the tree with each newly-added node is evident in the depicted example. It is understood that the given example is a single planning scenario. As such, statistical analysis of results obtained from a broader range of planning scenarios using this algorithm are presented in Section 5.6. The purpose of those results is to compare the performance of this path planner to that of the adapted anytime RRT of Section 5.4.2. The results will also serve as a baseline for measuring the effectiveness of the path-optimisation step when incorporated in the continuous-curvature anytime RRT and informed RRT* algorithms.

5.4.4 Summary: Adaptation of the Anytime RRT and the Informed RRT* to the Continuous-curvature Case

This section presented the development of the basic RRT, anytime RRT and informed RRT* path-planning algorithms, adapted for the planning of continuous-curvature paths. The continuous-curvature basic RRT algorithm was developed for the purpose of laying a foundation for the development of the latter two algorithms. The paths planned by the developed algorithms can be readily executed by vehicles with a constrained angular velocity and angular acceleration. Example path-planning results were shown to demonstrate the success of the implementation of the developed algorithms – the algorithms were implemented in MATLAB and C++, with MATLAB used for fast prototyping and testing, while the C++ implementation will be used in deployment of the developed algorithms in ROS for the demonstration of the integrated autonomous navigation system.

For each developed algorithm, the presented example path planning results were for a single path-planning scenario, with statistical results deferred to Section 5.6. The purpose of the latter results is two-fold: firstly, it is to analyse aggregated performance of each benchmark path-planning algorithm; secondly, they will be used as a baseline to evaluate the effectiveness of the path-optimisation algorithms of Section 5.5 in accelerating the convergence of the developed benchmark path planners.

While the paths produced by the implemented adaptations of the path-planning algorithms developed in this section can be readily executed by vehicles with a constrained angular velocity and angular acceleration, we are interested in investigating the effect of incorporating a path-optimisation step into each algorithm in accelerating convergence. For this purpose, the developed path-planning algorithms will be augmented with a path-optimisation step in Section 5.6 to form optimised path planners. The path-optimisation step will take the form of the path-optimisation algorithms developed in the next subsection. The effectiveness of each path-optimisation algorithm in accelerating the convergence of the benchmark path planners will be analysed.

5.5 Algorithm Development: Continuous-curvature Path-optimisation Algorithms

The paths generated by the continuous-curvature (CC) benchmark path-planning algorithms developed in the previous section, namely the CC basic RRT, the CC anytime RRT and the CC informed RRT* are composed of a continuous-curvature concatenation of primitive path segments, i.e.. clothoids, circular arcs and straight-line segments. Each path from the root to the goal can be represented as a list of n nodes: $\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_{n-1}$, with $\mathbf{q}_0 = \mathbf{q}_I$ and $\mathbf{q}_{n-1} = \mathbf{q}_G$. These are the vertices of the search tree that form the solution path. While these paths can be readily executed by vehicles with a constrained turning rate since they have a continuous-curvature profile, they

are either far from optimal (e.g. basic RRT) or approximate optimality can only be achieved at an additional computational cost (e.g. anytime RRT and informed RRT*). It is thus worthwhile to consider and investigate the use of a path optimiser alongside these algorithms, taking paths generated by each algorithm as an initialisation and improving them towards the optimal solution. The difference in the quality of paths returned by each algorithm serves as a wide range of possible initialisations, which is good for testing the effectiveness of the path optimiser. The review of path-optimisation techniques was the subject of Section 2.8. Among the reviewed techniques, shortcut-based path optimisation and gradient-based path optimisation were selected for use in this project. This section is dedicated to the development of the selected path-optimisation algorithms. The constraint placed on each developed path-optimisation algorithm is that in optimising an input continuous-curvature path, the output optimised path should also be curvature continuous – i.e. curvature continuity should be preserved. The developed path optimisers are later incorporated into the benchmark path-planning algorithms that were developed in the preceding chapter in Section 5.6.

5.5.1 Continuous-curvature Shortcut-based Path Optimisation

This subsection focuses on the development of shortcut-based path-optimisation algorithms. As explained in Section 2.8.1, shortcut-based path-optimisation techniques shorten an input node path, represented by a sequence of n nodes $\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_{n-1}$, by bypassing redundant nodes where possible.

In Section 2.8.1, existing path-optimisation algorithms based on the application of shortcuts to an input node path were reviewed. These are namely: path pruning, the random shortcut method and the wrapping process. Path pruning was presented as the simplest form of shortcut-based path optimisation, which considers a node, \mathbf{q}_{i+1} , on the node path as redundant if there exists a collision-free path from \mathbf{q}_i to \mathbf{q}_{i+2} . The path-pruning algorithm thus simply iterates through the nodes of the node path, starting from the first node, \mathbf{q}_0 , until the last node, \mathbf{q}_{n-1} , is reached, attempting these shortcuts and removing redundant nodes where shortcuts succeed. The algorithm is easy to implement. An issue with it is its dependency on the possibility of connecting at least one pair-of non-consecutive nodes if it is to have any chance of shortening the path. More precisely, with no connectible non-consecutive nodes, the technique fails, even if it would be otherwise possible to shorten the path. The shortcut method addresses this issue with path pruning, removing the dependency on the existence of at least one pair of connectible non-consecutive path nodes. It does so by not only using the nodes of the node path, $\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_{n-1}$, to shorten the path through shortcuts, but instead, discretising the node path and then using the points introduced by this discretisation on the node path to attempt shortcuts. The result is that even if initially, among the nodes of the input node path, no connectible non-consecutive nodes were connectible – hence no possible shortcuts, the discretisation may make it possible for the new shortcuts to be possible. The downside of this shortcut method is that the discretisation may introduce many new points on the path, which results in a large number of shortcuts to be attempted and consequently may take longer than the path-pruning method. Lastly, the wrapping process makes use of path discretisation and short-cutting to shorten paths in a manner that causes the optimised path to wrap around obstacles. The development of these three shortcut-based path-optimisation algorithms is the subject of this subsection. Due to the requirement for the path optimiser to preserve curvature continuity when shortening an input path, it is required for each shortcut applied by the shortcut-based path-optimisation algorithms developed in this subsection to be curvature continuous and for its concatenation with the rest of the path to also be curvature continuous. This is where the local path-planning method (LPM) developed in Chapter 4 comes in – given two configurations on the original path between which a shortcut is to be attempted, a call to the LPM with these two configurations as inputs generates a continuous-curvature local path satisfying the curvature at both the start and end configuration while also being curvature continuous everywhere. The next subsection focuses on the development of the path-pruning algorithm adapted for continuous-curvature path optimisation. The random shortcut method and the wrapping process, both adapted for curvature continuity are developed in Sections 5.5.1.2 and 5.5.1.3.

5.5.1.1 Continuous-curvature Path-pruning Algorithm

We now present the development of our adapted version of the path-pruning algorithm. Again, as explained and illustrated in Subsection 2.8.1, this path-optimisation algorithm works by iterating through the input node path, starting from the first node, $\mathbf{q}_0 = \mathbf{q}_I$, proceeding forward until the node that is third from last is reached, is reached; at each \mathbf{q}_i the algorithms checks if the next node, \mathbf{q}_{i+1} , is a redundant node, and if so, it replaces the sub-path from \mathbf{q}_i to \mathbf{q}_{i+2} via \mathbf{q}_{i+1} with a direct path (shortcut) from \mathbf{q}_i to \mathbf{q}_{i+2} . Pseudocode for the algorithm is given in Algorithm 23.

Algorithm 23: ccPathPruning(node path \mathbf{Q})

```

1  $i \leftarrow 0$  ; // Start from the first node of the node path,  $\mathbf{q}_0$ 
2 while  $i < |\mathbf{Q}| - 2$  do
3    $\text{localPath} \leftarrow \text{LPM}(\mathbf{q}_i, \mathbf{q}_{i+2})$  ; // Computation of shorter replacement path.
4   if  $\text{collisionFree}(\text{localPath})$  then // Shorter replacement path is collision-free.
5      $\mathbf{Q} \leftarrow \mathbf{Q} \setminus \mathbf{q}_{i+1}$  ; // Remove redundant node from node path.
6     if  $i > 0$  then
7        $i \leftarrow i - 1$  ; // After a successful shortcut, step back by one node.
8   else
9      $i \leftarrow i + 1$  ; // Proceed to next node if shortcut is not possible.
10 return  $\mathbf{Q}$ 

```

The adapted version presented in this subsection works exactly the same way as original version described in the literature review, except the adaptation for continuous-curvature path optimisation by ensuring that generated shortcuts are curvature continuous. The algorithm receives a node path \mathbf{Q} as an input. This node path is again represented by its path nodes, $\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_{n-1}$. Starting from the first node, \mathbf{q}_0 , the algorithm iterates through path nodes until the node that is third from last is reached (lines 1–2). At each iteration, i , the algorithm attempts a shortcut from the current node, \mathbf{q}_i , to node \mathbf{q}_{i+2} , bypassing node \mathbf{q}_{i+1} . The shortcut attempt is done through a call to the local path planner and the collision detector – the continuous-curvature local path planner computes the shorter replacement path (line 3) and the collision detector is used to examine the replacement path for the absence of collisions (line 4). On success of the shortcut attempt, the redundant node, \mathbf{q}_{i+1} , is removed from the node path (line 5). The success of a shortcut attempt means that the number of nodes on the node path reduces by one (lines 6–7). An unsuccessful shortcut attempt simply means the node \mathbf{q}_{i+1} is not redundant and results in the algorithm proceeding to check the next node, \mathbf{q}_{i+2} , for redundancy (lines 8–9). Finally, having attempted all possible shortcuts, the algorithm terminates and returns the optimised node path, with redundant path nodes removed (line 10).

The described continuous-curvature path-pruning algorithm has been written in MATLAB and C++. MATLAB has again been used for fast prototyping and testing while the C++ implementation is used later in the deployment of the algorithm through ROS during the demonstration of the integrated autonomous navigation system. Figure 5.6 gives an illustration of our MATLAB implementation of the developed algorithm in action. Paths generated by the continuous-curvature basic RRT are given as inputs to the path optimiser. The results of the optimisation are paths with their lengths certainly reduced. It is important to note that although in this particular example, the demonstration specifically considered a path produced by the CC basic RRT, the algorithm also applies to paths produced by the other continuous-curvature path-planning algorithms, namely the CC anytime RRT and the CC informed RRT*. In fact, the algorithm can be applied to paths generated by any path planner, as long as such paths are both curvature continuous and well presented as a node path.

With the continuous-curvature path-pruning algorithm developed and demonstrated, the next subsection presents the development of the second shortcut-based path-optimisation algorithm, the random shortcut method.

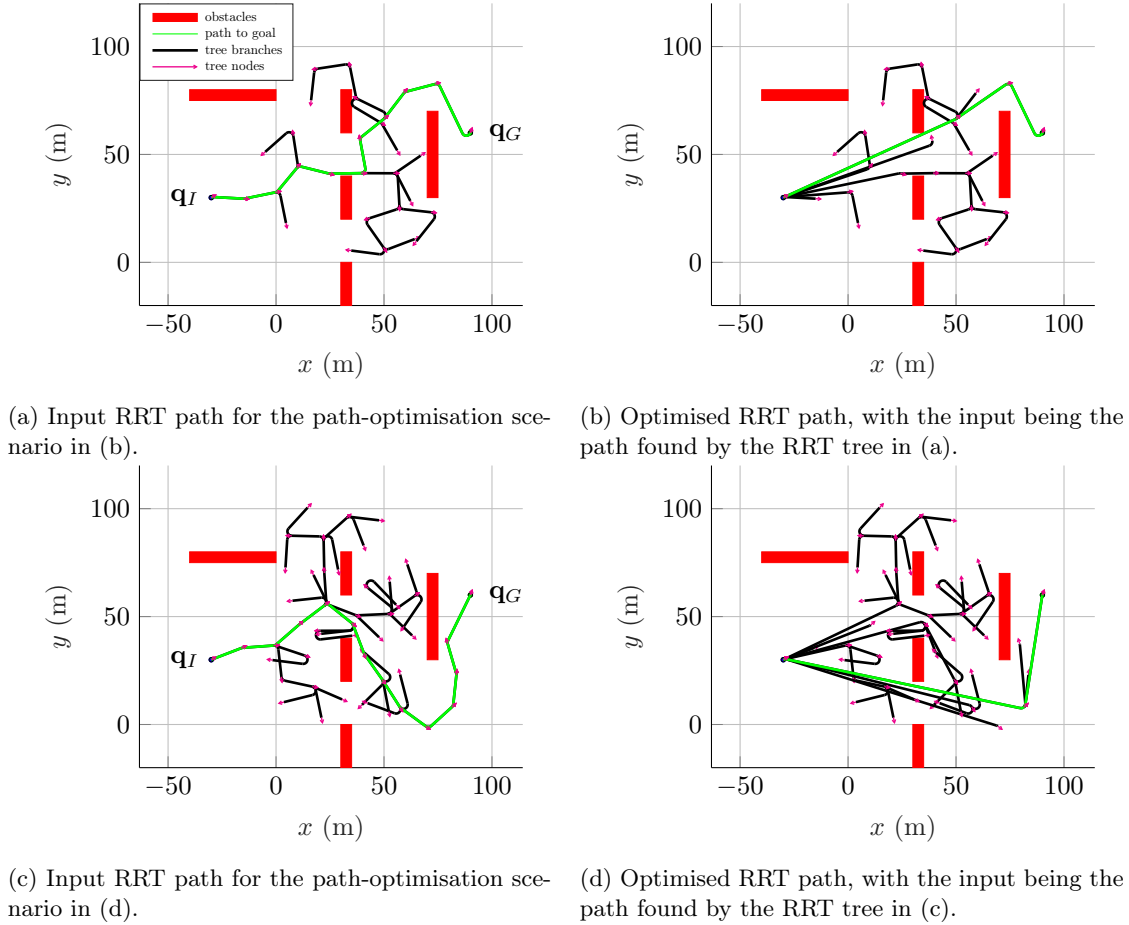


Figure 5.6: Examples illustrating the application of the continuous-curvature path-pruning algorithm to shorten a path generated by the continuous-curvature basic RRT algorithm.

5.5.1.2 Continuous-curvature Random-shortcut Method

The random-shortcut method was presented as an improvement to the path-pruning algorithm whose continuous-curvature version was developed and illustrated in the previous subsection. Particularly, instead of simply attempting shortcuts on the original nodes of the node path being optimised like path pruning does, the random-shortcut method discretises the input node path before starting to attempt shortening it. It then uses the configurations produced by this discretisation in its quest to shorten the path. The method works by iteratively splitting the path into three parts, then attempting to replace the middle part of the split path with a shorter replacement path. The version of the method developed in this subsection is adapted for curvature continuity through the use of the continuous-curvature LPM developed in Chapter 4.

Pseudocode for the algorithm is shown in Algorithm 24. The algorithm receives the discretised node path, \mathbf{Q} , as an input. At each iteration, it starts by picking two random configurations, \mathbf{q}_a and \mathbf{q}_b , that are then used to split the path into 3 parts (lines 3–7). With the path split, the algorithm attempts a shortcut for replacing the middle part of the path with a shorter path through the use of the continuous-curvature LPM (line 8). With this attempted shortcut collision-free, the middle portion of the path is replaced with the shorter, collision-free replacement path (lines 9–10). This process is repeated until the number of allocated iterations elapses, at which point, the shortened path is returned to the calling function.

The developed random-shortcut algorithm has been written in MATLAB and C++. In Figure 5.7, we illustrate its working by showing runs of our MATLAB implementation in optimising paths generated by the CC basic RRT. The depicted scenarios show a clear reduction in path length. It can be seen from the presented results that the random-shortcut algorithm introduces

Algorithm 24: ccRandomShortcut(discretised node path \mathbf{Q})

```

1  $K \leftarrow$  total number of iterations;
2 for  $i = 1$  to  $K$  do
3    $n \leftarrow |\mathbf{Q}|$  ; // number of configurations on discretised node path
4    $[a, b] \leftarrow a_{\text{rand}}, b_{\text{rand}}$  ; // 2 configurations that split path into 3 parts.
5    $\mathbf{Q}' \leftarrow \mathbf{q}_0, \dots, \mathbf{q}_{a-1}$  ; // the first part of the split path.
6    $\mathbf{Q}'' \leftarrow \mathbf{q}_a, \dots, \mathbf{q}_b$  ; // the second (middle) part of the split path.
7    $\mathbf{Q}''' \leftarrow \mathbf{q}_{b+1}, \dots, \mathbf{q}_{n-1}$  ; // the third part of the split path.
8   localPath  $\leftarrow$  LPM( $\mathbf{q}_a, \mathbf{q}_b$ ) ; // Local path computation.
9   if collisionFree(localPath) then
10     $\mathbf{Q} \leftarrow \mathbf{Q}' \cup \text{localPath} \cup \mathbf{Q}'''$  ; // Middle portion replaced with shorter path.
11 return  $\mathbf{Q}$ ;
```

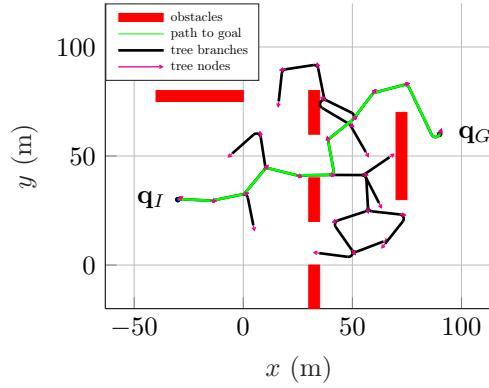
new configurations to the input node path prior to attempting random shortcuts to shorten it. The advantage that comes with introducing new nodes as opposed to simply attempting shortcuts using the original path nodes (as in path pruning) can be seen in the first example (Figures 5.7(a) and 5.7(b)) where random shortcuts to newly inserted nodes result in path length reduction that would have otherwise been impossible without the introduction of the additional nodes (see upper right portion of the optimised path of Figure 5.7(b)). In the presented results, the algorithm was allocated 10 iterations. At this point, the number of iterations of the algorithm was simply set to 10 for the purpose of demonstrating the working without necessarily warranting that this number of iterations is sufficient in the general case.

We have now developed and demonstrated a continuous-curvature version of the random-shortcut path-optimisation algorithm. We now move on to the third shortcut-based path-optimisation algorithm developed in this thesis, the continuous-curvature wrapping process.

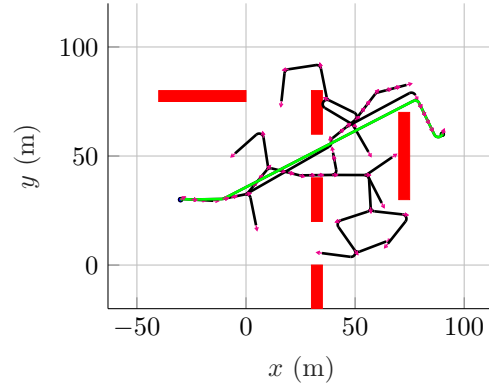
5.5.1.3 Continuous-curvature Wrapping Process

The wrapping process is the last shortcut-based path-optimisation algorithm developed in this thesis. As explained in Subsection 2.6.4.4, the technique works by combining the principles of path discretisation and short-cutting to shorten an initial node paths in a manner that causes the optimised path to wrap around obstacles. Its pseudocode is given in Algorithm 25.

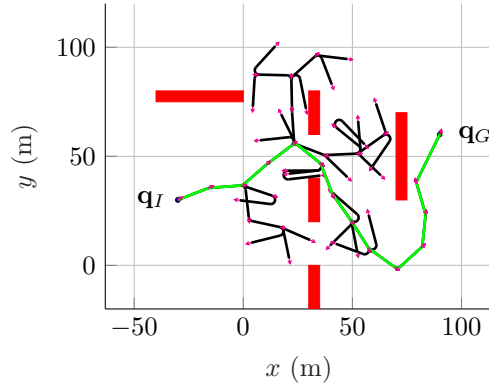
Like the path-pruning algorithm, the wrapping process takes a node path generated by a global path planner as an input. The algorithm then starts from the initial configuration, which is at the first index of the node path, and iterates until the node that is third from last on the node path is reached. For each indexed node (i.e. for the i^{th} node, \mathbf{q}_i), the algorithm discretises the path between \mathbf{q}_{i+1} and \mathbf{q}_{i+2} according to a discretisation parameter, D_1 . Then, starting from \mathbf{q}_{i+2} and moving backwards towards \mathbf{q}_{i+1} in steps of the discretisation parameter, the algorithm computes a temporary configuration, \mathbf{q}_{temp} (line 4), and attempts to connect configuration \mathbf{q}_i directly to \mathbf{q}_{temp} with a collision-free path (line 5). The configuration \mathbf{q}_{i+1} is pruned from the node path if it is possible to exactly connect \mathbf{q}_i to $\mathbf{q}_{\text{temp}} = \mathbf{q}_{i+2}$ (lines 6–9). Otherwise, the algorithm starts from \mathbf{q}_i and moves towards \mathbf{q}_{temp} in steps of a second discretisation parameter, D_2 , computing a new configuration, \mathbf{q}_{new} (line 13). For each computed new configuration, the algorithm attempts to make a collision-free connection from \mathbf{q}_{new} to \mathbf{q}_{i+2} . The first configuration from which the collision-free connection to \mathbf{q}_{i+2} succeeds is used as a cheaper replacement for \mathbf{q}_{i+1} (line 13–16). This process repeats until the configuration that is third from last is reached, at which point a modified path that wraps around the obstacle is obtained. The difference between the version of the algorithm developed in this thesis and that described in literature is that the one developed here is adapted for the optimisation of continuous-curvature paths while the one described in literature produces paths with a discontinuous-curvature profile. The adaptation for continuous-curvature path optimisation has been realised by making use of the continuous-curvature local path planner developed in Chapter 4. An additional consideration taken into account in the version developed in this project, which is not an issue in the discontinuous-curvature version described in literature,



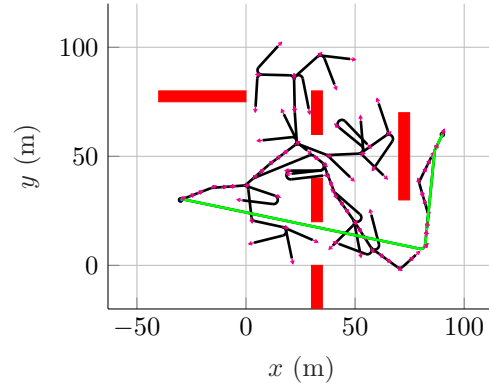
(a) Input RRT path for the path optimisation scenario in (b).



(b) Optimised RRT path, with the input being the path found by the RRT tree in (a).



(c) Input RRT path for the path optimisation scenario in (d).



(d) Optimised RRT path, with the input being the path found by the RRT tree in (c).

Figure 5.7: Examples illustrating the application of the continuous-curvature random-shortcut algorithm to shorten a path generated by the continuous-curvature basic RRT algorithm.

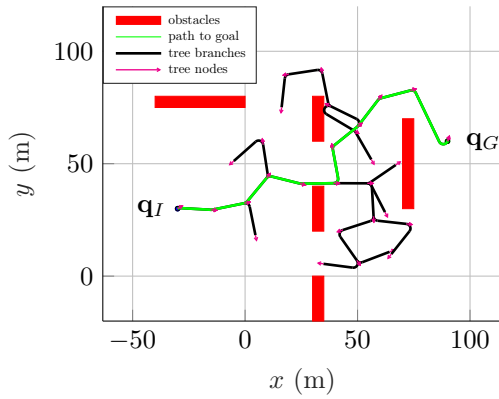
is that it is important to ensure that the configurations inserted on the path during discretisation satisfy the orientation and curvature constraints of the planned path.

The developed continuous-curvature wrapping process has been written in MATLAB and C++. We demonstrate it in action by showing runs of the MATLAB implementation in optimising paths generated by the basic RRT. The effect of the wrapping process, i.e. shortening the input path in such a manner that the optimised path around obstacles, is evident in the presented results. This results in paths that are shorter than those found by the path-pruning and random-shortcut algorithms. It should be noted though that, in principle, the random-shortcut algorithm is capable of producing paths that wrap around obstacles if given enough iterations. The continuous-curvature wrapping process is the last shortcut-based path-optimisation algorithm developed in this thesis. The next subsection summarises the outcomes of the development of the three shortcut-based path-optimisation algorithms discussed above.

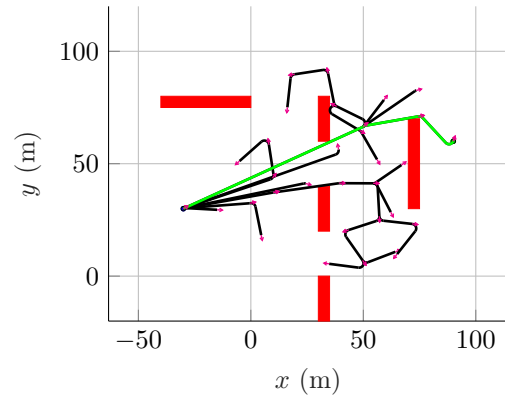
Algorithm 25: wrap(node path Q)

```

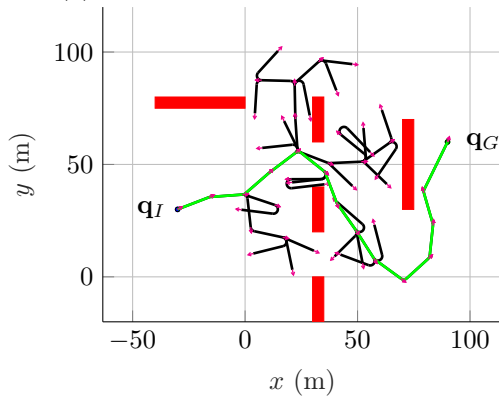
1   $i \leftarrow 1$ ;
2  while  $i < |Q| - 2$  do
3      for  $k = 0$  to  $D_1 - 1$  do
4           $q_{temp} \leftarrow q_{i+2} + (q_{i+1} - q_{i+2}) \cdot \frac{k}{D_1}$  ; // Discrete step towards  $q_{i+1}$  from  $q_{i+2}$ .
5          if collisionFree(LPM( $q_i, q_{temp}$ )) then
6              if  $k = 0$  then
7                   $N \leftarrow N \setminus q_{i+1}$  ; // Prune  $q_{i+1}$  from node path.
8                   $i \leftarrow i - 1$ ;
9                  break;
10             else
11                 for  $m = 1$  to  $D_2 - 1$  do
12                      $q_{new} \leftarrow q_i + (q_{temp} - q_i) \cdot \frac{m}{D_2}$  ; /* Discrete step towards  $q_{temp}$  from  $q_i$ . */
13                     if collisionFree(LPM( $q_{new}, q_{i+2}$ )) then
14                          $q_{i+1} \leftarrow q_{new}$  ; /* Replace  $q_{i+1}$  with a cheaper configuration. */
15                         break;
16                     if  $m == D_2 - 1$  then
17                          $q_{i+1} \leftarrow q_{temp}$  ; /* Replace  $q_{i+1}$  with a cheaper configuration. */
18                         break;
19              $i \leftarrow i + 1$ ;
    
```



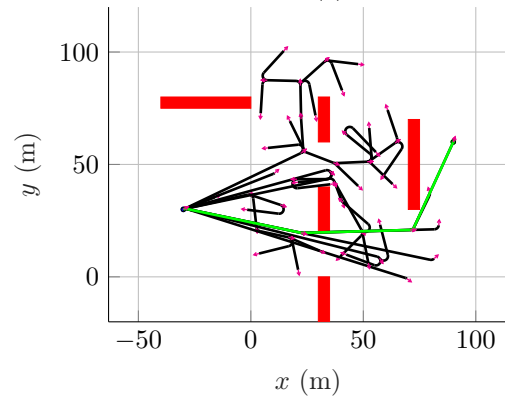
(a) Input RRT path for the path-optimisation scenario in (b).



(b) Optimised RRT path, with the input being the path found by the RRT tree in (a).



(c) Input RRT path for the path-optimisation scenario in (d).



(d) Optimised RRT path, with the input being the path found by the RRT tree in (c).

Figure 5.8: Examples illustrating the application of the continuous-curvature wrapping process to shorten a path generated by the continuous-curvature basic RRT algorithm.

5.5.1.4 Summary: Developed Shortcut-based Continuous Curvature Path Optimisation Algorithms

At this point, three continuous-curvature shortcut-based path-optimisation algorithms have been developed and demonstrated. These are continuous-curvature path pruning, the continuous-curvature random shortcut and the continuous-curvature wrapping process. These path-optimisation algorithms are now ready to be incorporated into the path-planning algorithms developed in Section 5.4 so as to investigate their utility in accelerating the rate of convergence of these path planners towards the optimal solution. The following subsection develops a second class of path-optimisation algorithms, namely those which are gradient-based. They will also be used for the same purpose as the developed shortcut-based path optimisers and their utility in accelerating the rate of convergence of the developed path planners will be compared to that of their shortcut-based counterparts.

5.5.2 Continuous-curvature Gradient-based Path Optimisation

The literature review for path optimisation presented in Section 2.8 identified two categories of path-optimisation techniques, namely those based on shortcuts and those based on numerical optimisation. Furthermore, existing algorithms in each category were reviewed, compared and suitable ones selected for adaptation and extension in this project. The previous subsection presented the development and extension of shortcut-based path-optimisation algorithms. This subsection focuses on the development of the algorithm that was identified as suitable for our use under numerical path-optimisation techniques, this is gradient-based path optimisation.

The gradient-based path-optimisation algorithm that has been selected for adaptation in this project is Campana et al.'s algorithm [136], which was detailed in Subsection 2.8.2. It is again shown in Algorithm 26 for convenience. What made the algorithm relevant to our work is that it directly optimises path length, which is the same path-optimisation objective as that of this project. Again, the difference is that their algorithm optimises paths that are a concatenation of straight lines, while in this project we are interested in continuous-curvature paths. It is therefore necessary to consider some aspects of the algorithm and how they are adapted for our use.

Algorithm 26: gradientBasedPathOptimisation(node path \mathbf{Q})

```

1  $\alpha_i \leftarrow \alpha_{\text{init}}$  ; // Initialise descent step length
2 minReached  $\leftarrow$  false ; // No solution yet
3 while not(noCollision and minReached) do
4    $\mathbf{P} \leftarrow \text{computeOptimalStep}()$  ; // LCQP optimal step
5   minReached  $\leftarrow$  ( $\|\mathbf{P}\| < 10^{-3}$  or  $\alpha_i == 1$ ) ; // Solution found
6    $\mathbf{Q}_{i+1} \leftarrow \mathbf{Q}_i + \alpha_i \mathbf{P}$  ; // New set of path nodes
7   if not(collisionFree( $\mathbf{Q}_{i+1}$ )) then
8     noCollision  $\leftarrow$  false ; // New solution contains collisions
9     if  $\alpha \neq 1$  then
10       computeCollisionConstraint( $\mathbf{Q}_{i+1}, \mathbf{Q}$ ) ; /* Compute collision constraint based
11         on new path and previous collision-free path */
12       findNewConstraint() ; /* Find constraint that is linearly independent */
13       addCollisionConstraint() ; /* Insert constraint to constraint Jacobian matrix */
14        $\alpha_i \leftarrow 1$  ; /* Revert to attempting to directly reach the minimum in one step */
15     else
16        $\alpha_i \leftarrow \alpha_{\text{init}}$  ; /* Revert to small steps if collisions have been detected */
17   else
18      $\mathbf{Q} \leftarrow \mathbf{Q}_{i+1}$  ; // A new, collision-free path has been found
19     noCollision  $\leftarrow$  true;
19 return  $\mathbf{Q}$ 

```

We start with the nature of the paths. In Campana et al.'s algorithm, with new path nodes (the set of configurations forming the path), \mathbf{Q}_{i+1} , computed (line 6) in a descent step, the re-

sulting path is simply a straight-line interpolation among the nodes (line 6). In our case, however, the configurations that form the path are not connected with straight lines, but rather with continuous-curvature local paths. This means that in our case it is necessary to iterate through the configurations that form the iterate with the continuous-curvature LPM that was developed in Chapter 4 before the newly-found path is checked for collisions. Then the path can be checked for collisions by performing collision detection on each of the resulting local paths. The new path is collision-free only if all these local paths are collision-free.

The use of the local path planner in our adapted version of the algorithm has an implication on the path length. More precisely, while in the case of a straight-line interpolation, the cost of the subpath connecting any two configurations on the path is always directly proportional to the separation between the configurations, this is not always the case in the case of a continuous-curvature local path. Particularly, when two consecutive configurations are too close to each other, the path connecting them is either self intersecting or purely made up of turns. As a result, the path length given by:

$$\|\mathbf{q}_{i+1} - \mathbf{q}_i\|_W \triangleq \sqrt{(\mathbf{q}_{i+1} - \mathbf{q}_i)^T W^2 (\mathbf{q}_{i+1} - \mathbf{q}_i)}, \quad (5.5)$$

that forms the basis of the cost function:

$$L(\mathbf{Q}) = \sum_{k=1}^{n-2} \|\mathbf{q}_k - \mathbf{q}_{k+1}\|_W^2, \quad (5.6)$$

would no longer be always relevant. To keep the cost function relevant, though as a *surrogate cost function*¹, we enforce a minimum separation between consecutive configurations that eliminates the occurrence of self-intersecting or pure-turn local paths.

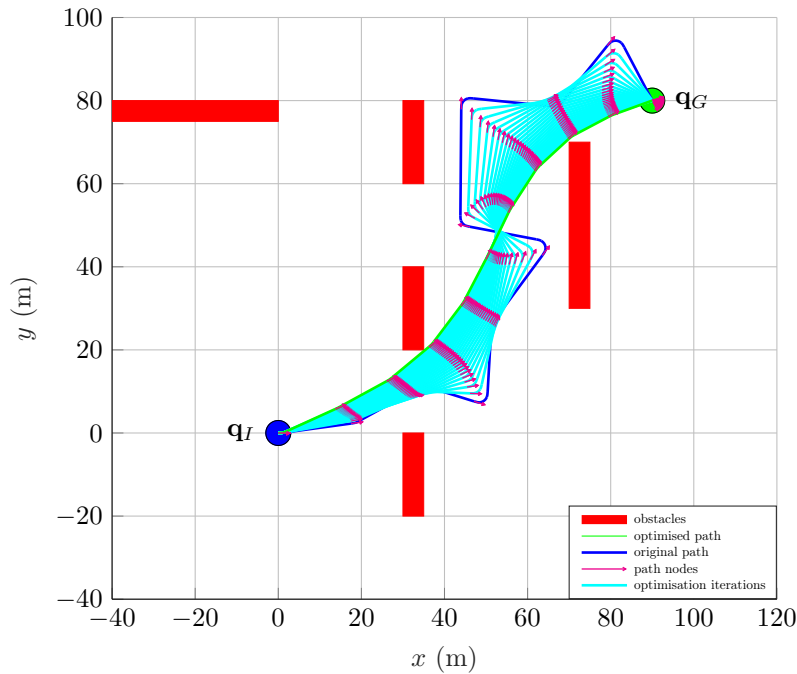


Figure 5.9: An example illustrating the application of the gradient-based path-optimisation algorithm to a path generated by the continuous-curvature basic RRT algorithm.

¹A surrogate cost function is a cost function which mimics properties of interest of the actual cost function, while being computationally inexpensive and possessing desirable analytic properties including convexity and ease of differentiability.

algorithms developed and discussed, the next section concludes this chapter on optimised path planning by incorporating path optimisation into the benchmark path-planning algorithms. The section also presents analyses and results for optimised global path planners, which are formed through this incorporation of the developed path-optimisation algorithms into the benchmark path planners.

5.6 Optimised Path Planning Results: Incorporating Path Optimisation into the Benchmark Path Planning Algorithms

This section discusses the incorporation of the path-optimisation algorithms developed in Section 5.5 into the benchmark path planners developed in Section 5.4. The benchmark path planners to be augmented with the developed path optimisers are the CC anytime RRT and the CC informed RRT*. As pointed out earlier, these path-planning algorithms work in a similar manner in that they both strive towards the optimal solution by using the cost of each newly-found solution to iteratively bound the sampling space – a process that encourages each subsequent search to find a solution with a lower cost than all previous ones. The difference between the algorithms is in the quality of the paths that they find and their execution time. Particularly, for the same sampling sequence, the CC informed RRT* finds low-cost paths compared to the CC anytime RRT, due to the presence of the rewiring process in the CC informed RRT* and the absence of the same in the CC anytime RRT. This advantage of the CC informed RRT* over the anytime RRT comes at the expense of computation time. Thus while the CC anytime RRT is less optimal than the CC informed RRT*, it is generally faster. As explained in Section 5.4.2, the CC anytime RRT has two extremes with regards to the paths it produces and its computation time. At one extreme, by simply considering only one neighbour for tree extension, it extends the search tree in exactly the same manner as the basic RRT and as such it is very fast under this setting, though producing costly solutions. At the other extreme, it selects the cheapest among a set of neighbours when extending the tree. Under this second setting, the algorithm is slower than in the first setting, but produces paths with lower costs. In this section, two versions of the CC anytime RRT are used, one configured so as to exhibit the first extreme and the other exhibiting the second extreme. We refer to the first version as the *CC basic anytime RRT*, and to the second one as the *CC k-nearest anytime RRT*. As a result, we have three benchmark path-planning algorithms, with the CC informed RRT* being the third.

The modification of the benchmark path-planning algorithms to include a path-optimisation step is straightforward. Each time a given benchmark path planner finds a path to the goal, this path is fed to the path optimiser, whose role is to reduce the cost of the input path. The optimised path is then used to bound the sampling space in the subsequent search. This process repeats until the planning period elapses. In this manner, the path optimiser serves as a catalyst to the planning process, accelerating its rate of convergence towards the optimal solution. The manner in which we incorporate the path-optimisation step has to be explained. There are two ways in which the path-optimisation step can be incorporated to the benchmark path planners. Firstly, it can be simply be applied to the path being optimised, without affecting any tree nodes. The advantage of this first approach is its simplicity since no changes (be it the addition of new nodes introduced during path optimisation or updates on affected existing ones) are effected on the tree. Secondly, the new nodes introduced in the course of the path-optimisation process can be added to the tree as path optimisation proceeds. Moreover, existing tree nodes that are connected to the path being optimised can have their cost-to-come reduced as a result of the optimisation process. This second approach has the effect of automatically improving the costs of descendent nodes of the nodes visited during path optimisation, even though they are not part of the path being optimised. For this reason, we use the second approach due to its ability to utilise the path-optimisation step not only to improve the solution path, but also to improve parts of the tree.

With the path-optimisation step incorporated to each of the benchmark path planners, we are ready to perform analyses of the effectiveness of each path-optimisation algorithm in accelerating the rate of convergence of each benchmark path planner. In the next subsection, we introduce our experimental approach.

5.6.1 Experimental Approach

The rest of this section presents analyses and comparisons of the developed optimised path planners. We begin with analyses and comparisons of the paths produced by the respective benchmark path planners in Subsection 5.6.2. This is an important step as these analyses and comparisons serve as a baseline for the optimised path planners, whose similar analyses and comparisons are presented in Subsection 5.6.3. The idea is that by comparing the baseline performance of a given global path planner against the performance obtained by applying each of the different path-optimisation algorithms to it, then fair judgements on the versatility of each path optimiser can be made. The conclusions drawn about the benchmark and optimised path-planning algorithms developed in this thesis are based on results obtained from 400 sample runs of each algorithm, with a maximum of 500 iterations allocated for each run. At this point, it is important to recall that, as demonstrated in the discussion of the results of the CC basic RRT (Section 5.4.1), due to the randomised nature of the benchmark path planners, there is also random variation in the costs of paths they find in different runs of the same path-planning query. As a result, if each of these experiments is repeated, results obtained will vary. This is because the results for each experiment are based on paths found in the selected number of runs of each algorithm in that experiment, not on all the possible paths each algorithm can find in the configuration space. For this reason, in the analyses of the results of each experiment, we need to include a measure of this variability. This is where *confidence intervals*² come into play. The measure of variability between results obtained from repetitions of each experiment is called the *margin of error*. It provides bounds on the extent by which the results of repetitions of each experiment can be expected to differ from our results – that is, bounds on the range within which our results for that experiment are accurate. The confidence interval is then that range. In other words, the range of likely values for the parameter being estimated. A narrow confidence interval implies that the results are more precise.

We now proceed to the analyses and comparison of the benchmark path planners.

5.6.2 Analyses and Comparison of the Benchmark Global Path Planners

Experimental results demonstrating the working of the three benchmark global path planners, namely the CC basic anytime RRT, the CC k-nearest anytime RRT and the CC informed RRT* have been already presented in the respective subsections of Section 5.4. From those results, it is possible to roughly see the difference in the cost, in terms of path length, of the paths produced by the respective benchmark path planners. Nevertheless, statistical analyses of their comparative performance are useful. As explained in the preceding section, statistical results are important since they give a bigger picture of how each algorithm can be expected to perform in general, as opposed to the limited view provided by results obtained from one-shot runs. In this subsection, we present statistical analyses and comparison of the three algorithms to establish their baseline head-to-head performance. This baseline will be then used to measure the effectiveness of incorporating the path-optimisation step in the optimised versions of the algorithms.

In line with our experimental approach, described in Subsection 5.6.1, 400 runs of each benchmark path planner have been performed. Figure 5.11 shows percentile plots that summarise the evolution of path costs as a function of the number of iterations for each benchmark path planner. The environment in which the path planning was performed for these results is the same environment used in illustrating the working of the algorithms in Sections 5.4.1, 5.4.2 and 5.4.3. As opposed to the experimental results used to demonstrate the working of the algorithms in those subsections, which showed one run for each algorithm, the results presented in this subsection show the evolution of the path cost over the range of allocated iterations in a population of paths found by 400 independent runs of each algorithm. In each run, each algorithm finds an initial path and then uses the remaining number of iterations attempting to improve it, possibly finding better paths in that run. The costs of the solutions found in each run represent the evolution of the path cost within that run. This information is then presented in percentiles, along with the minimum and the maximum costs, as shown in Figure 5.11. It should be noted that it is possible that at a given iteration of a path planner run, a feasible solution may not have been found yet. This is particularly likely for iterations that are early in the planning process, where the search tree has

²We detail confidence intervals in Appendix A.1

not yet reached the goal. For such an iteration, we consider the path to have an infinite cost, since it has not been found yet and as such its cost is unknown.

Some differences in performance between the three benchmark path planners can be observed visually, by inspection of the percentile plots. Others, which cannot be easily read off the plots, are better quantified numerically. For this reason, a numerical summary of some additional statistics of the benchmark path planners is given in Table 5.1. Due to space constraints in this document, these numerical statistics are not provided for every iteration of each algorithm, but rather for every 50th iteration.

Through visual inspection of Figure 5.11, we observe that the results verify our expectation and knowledge, and that recorded in literature about the expected head-to-head performance of the algorithms. The performance of our adapted versions of the respective algorithms for continuous-curvature path planning is indeed consistent with the performance of the original algorithms, which were based on simple straight-line local path planners in the respective literature. The CC informed RRT* performs best in improving the solution as shown by the highest decay in path cost, as the number of iterations increases. The CC k-nearest anytime RRT comes second in this regard, and the CC basic anytime RRT comes last. The CC informed RRT* is also able to find paths that are shorter, as demonstrated by plots of the minimum costs of the algorithms. In this regard, the CC k-nearest anytime RRT comes close to the CC informed RRT*, with its minimum being only 0.052% to 0.770% larger than that of the informed RRT* in most iterations. On the other hand, the minimum of the CC basic anytime RRT is significantly higher than those of the CC k-nearest anytime RRT and the CC informed RRT*, even noticeable by inspection; numerically it is up to 8.436% larger than the CC k-nearest anytime RRT's minimum and up to 8.920% larger than the informed RRT*'s minimum. The same trend is observed for the maximum cost of paths returned by the algorithms. The CC informed RRT* returns solutions with a lower maximum cost compared to the two other algorithms. The CC k-nearest anytime RRT comes second, again getting close to the CC informed RRT*, with its maximum being only 2.186% to 7.352% larger than the CC informed RRT*'s maximum. This is in contrast to the CC basic anytime RRT's maximum cost, which is significantly higher than the maximum costs of the other two algorithms. It is at least 29.177% larger than the CC k-nearest anytime RRT's maximum and at least 35.029% larger than the informed RRT*'s maximum. The CC informed RRT*'s maximum can be seen rapidly decreasing, approaching the minimum and narrowing the range of path costs, as the number of iterations increases. This is in line with the algorithm's notion of asymptotic optimality, i.e. that as the number of iterations tends to infinity, the probability that it finds the optimal solution tends to 1. The other two algorithms lack this asymptotic optimality guarantee.

As seen in Figure 5.11 and confirmed by the numerical statistics of Table 5.1, the same trend observed from both the minimum and maximum costs of solutions returned by the benchmark algorithm is also observed on all percentiles: the costs of all percentiles of the CC informed RRT* are consistently below those of the same percentiles in the other two algorithms. The CC k-nearest anytime RRT again comes second, closely contesting the CC informed RRT*, while the CC basic anytime RRT again lags behind by a large margin. The percentile plots suggest that the 75th percentiles of both the CC k-nearest anytime RRT and the CC informed RRT* have path costs that are below the costs of the 25th percentile of the CC basic anytime RRT. This is supported by numerical summary of the statistics presented in Table 5.1. Moreover, in later iterations (above 150 iterations for the CC k-nearest anytime RRT and above 100 iterations for the CC informed RRT*), the 95th percentiles of these two algorithms have path costs that are below the costs of the 25th percentile of the CC basic anytime RRT. Furthermore, in iterations beyond 200, the maximum cost of solutions returned by the CC informed RRT* is lower than the CC basic anytime RRT's 25th percentile.

The mean path cost at each iteration has been estimated, with 95% confidence. It should again be noted that it is possible that at some iterations of a given path planner run, a path may not have been found yet and as a result the corresponding path cost is considered infinite. A decision on how to handle such cases when computing the mean path had to be taken. Since it does not make sense to compute a mean for a dataset that possibly has infinite values, it was decided to exclude runs for which a path has not been found yet at each iteration. This principle is also applied in the computation of the standard deviation. There are two options for performing this exclusion of infinite-cost paths in computing the mean path cost and standard deviation per iteration. One

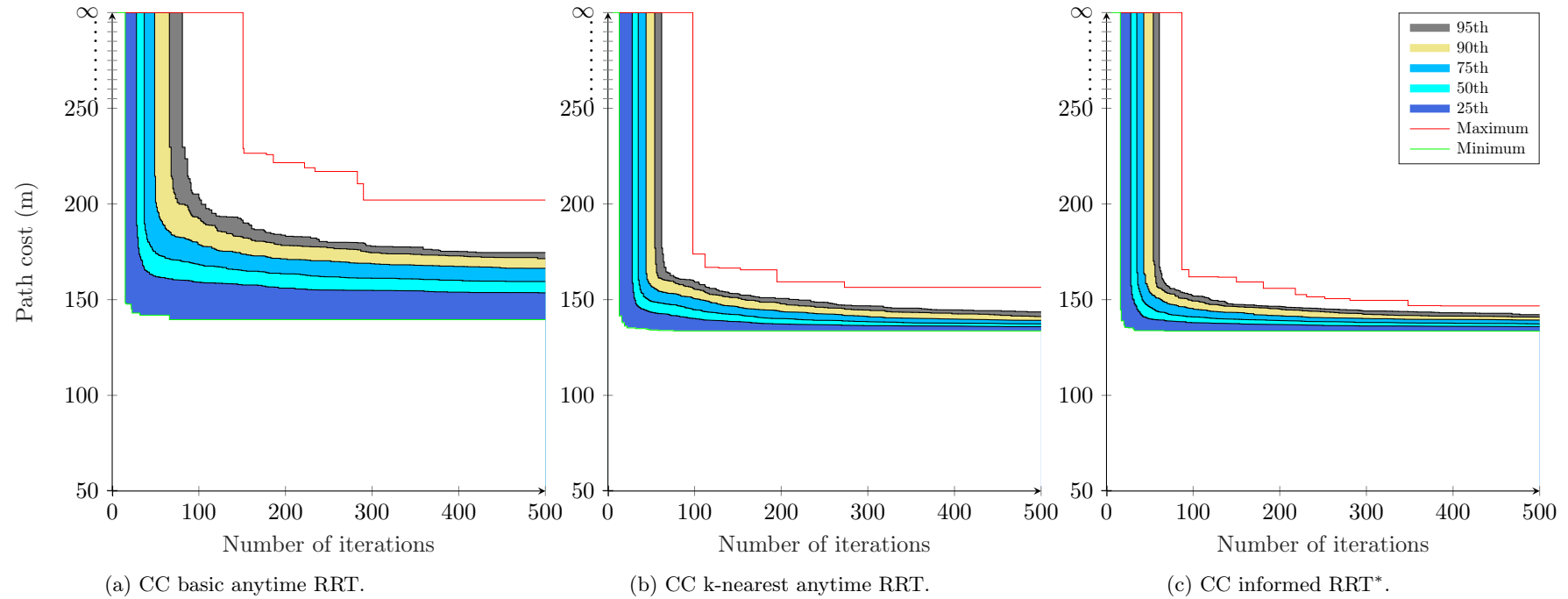


Figure 5.11: A percentile plot showing the evolution or improvement of path cost over an increasing number of iterations for 400 independent runs of each of the three benchmark global path-planning algorithms. The CC informed RRT* exhibits the highest cost decay with increasing number of iterations, the CC k-nearest anytime RRT comes second in this regard and the CC basic anytime RRT comes last. This is in line with the expected head-to-head performance of the algorithms.

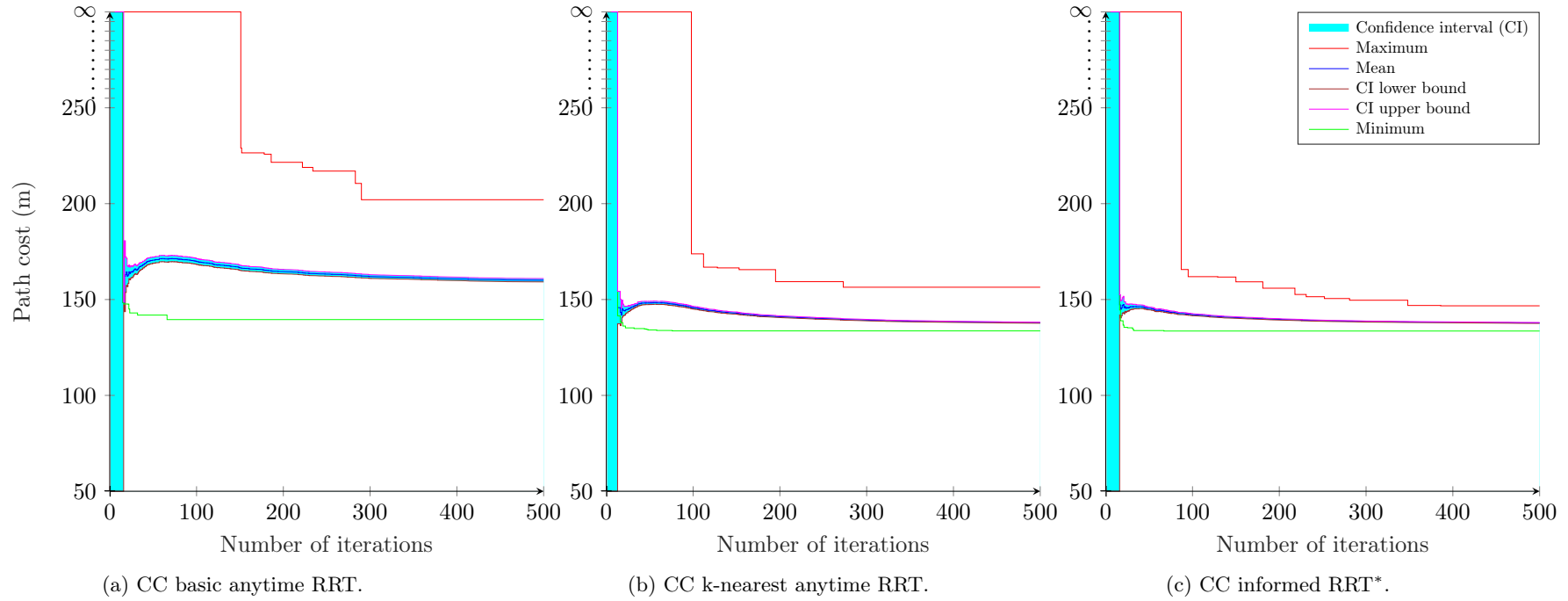


Figure 5.12: A plot of the mean path cost per iteration, with a confidence interval, for 400 independent runs of each benchmark global path-planning algorithm. The mean is calculated using only the finite-cost paths; see text for details.

	Iteration No.	Min.	Max.	\mathbf{q}_1	\mathbf{q}_2	\mathbf{q}_3	IQR	90 th	95 th	Mean	Std. dev.
CC basic anytime RRT	50	141.885	∞	161.933	174.150	200.718	38.785	∞	∞	170.540	14.764
	100	139.535	∞	159.130	168.337	178.791	19.661	192.326	203.072	169.745	15.494
	150	139.535	∞	157.654	165.253	173.831	16.177	182.795	191.565	166.489	13.782
	200	139.535	221.568	156.014	163.456	171.264	15.250	178.307	183.113	164.488	11.751
	250	139.535	216.990	155.024	161.928	170.323	15.299	177.098	179.959	163.242	10.922
	300	139.535	202.049	154.860	161.169	168.755	13.895	174.517	177.901	161.979	9.703
	350	139.535	202.049	154.583	160.782	167.948	13.365	173.627	177.421	161.392	9.457
	400	139.535	202.049	153.901	160.127	167.337	13.436	172.880	175.176	160.807	9.142
	450	139.535	202.049	153.723	159.638	166.547	12.824	172.028	174.602	160.281	8.867
	500	139.535	202.049	153.559	159.456	166.363	12.804	171.397	174.602	160.056	8.843
CC k-nearest anytime RRT	50	134.116	∞	143.370	149.073	156.842	13.472	∞	∞	148.251	7.849
	100	133.634	173.827	140.147	144.737	150.527	10.38	155.331	159.172	145.703	7.199
	150	133.634	166.492	138.541	142.135	146.067	7.526	150.825	153.019	142.895	5.667
	200	133.634	159.314	137.289	140.113	143.793	6.504	147.778	150.622	140.983	4.783
	250	133.634	159.314	136.894	139.182	142.474	5.580	145.843	148.589	140.072	4.264
	300	133.634	156.412	136.418	138.444	141.114	4.696	144.565	146.765	139.255	3.782
	350	133.634	156.412	136.255	137.858	140.259	4.004	143.372	145.554	138.679	3.316
	400	133.634	156.412	136.171	137.732	139.828	3.657	142.575	144.652	138.374	3.119
	450	133.634	156.412	135.993	137.514	139.401	3.408	141.989	144.257	138.105	2.982
	500	133.634	156.412	135.897	137.343	139.181	3.284	141.161	143.586	137.915	2.866
CC informed RRT*	50	133.816	∞	139.933	146.039	153.625	13.692	∞	∞	145.310	7.051
	100	133.564	161.922	137.963	141.001	145.084	7.121	149.255	151.869	142.048	5.238
	150	133.564	159.235	137.495	139.735	143.297	5.802	146.353	147.482	140.591	4.127
	200	133.564	155.905	136.936	139.064	141.586	4.650	145.043	146.440	139.641	3.577
	250	133.564	151.463	136.493	138.480	140.507	4.014	143.742	145.227	138.935	3.174
	300	133.564	149.634	136.226	138.064	140.118	3.892	142.332	144.084	138.497	2.840
	350	133.564	146.898	136.187	137.834	139.696	3.509	141.720	143.721	138.255	2.675
	400	133.564	146.720	136.057	137.710	139.641	3.584	141.384	143.139	138.062	2.565
	450	133.564	146.720	135.997	137.537	139.506	3.509	141.164	142.866	137.922	2.522
	500	133.564	146.720	135.947	137.315	139.314	3.367	140.894	142.148	137.791	2.435

Table 5.1: Numerical summary of the statistics of the benchmark path-planning algorithms at selected number of iterations. \mathbf{q}_1 , \mathbf{q}_2 , \mathbf{q}_3 are the 25th, 50th, 75th percentiles. The interquartile range (IQR) as well as the 90th and 95th percentiles are also shown

option is to only compute and report the mean and standard deviation for iterations in which all runs have found a path. Under this option, the mean and standard deviation are computed over all runs of the given path-planning algorithm. The other option is to compute and report the mean and standard deviation for all iterations, but only take into account finite-cost paths in each iteration. Under this option, at each iteration, the mean and standard deviation are only computed over runs that were able to find a feasible path. As a result, the mean and standard deviation have to be interpreted while bearing in mind which of these options were used to compute them. The standard interpretation of mean and standard deviation applies to the first option, but for the second option, the subset of runs represented must be understood. We elected to use the second option.

The estimated mean, with confidence bounds, for each benchmark path planner is plotted in Figure 5.12. A summary of the mean for every 50th iteration is also given in Table 5.1. Due to the fact that, at each iteration, the mean is computed using only the runs with finite-cost paths (as opposed to all runs), the plot of the estimated mean cannot be guaranteed to monotonically decrease; a monotonic decrease can only be guaranteed from the first iteration in which all runs have finite path costs. This is evident in Figure 5.12. For all the algorithms, with no path found in all runs, the mean path cost is infinite. Then, with at least one path found, the mean drops from the infinite value to a finite mean, computed over the runs that were able to find paths. Since the mean is not computed over all runs, it is not guaranteed to monotonically decrease, as long as there are some runs with infinite-cost paths. As seen in Figure 5.12, due to this reason, the estimated mean initially increases as the number of iterations increases. Later on, once all runs have finite-cost paths, the estimated mean decreases monotonically. The confidence interval starts off unbounded, when all runs have not found a path. It then decreases progressively with each iteration as more paths are found. The estimated means of the benchmark path planners are consistent with the other presented statistics. The CC informed RRT* has the lowest mean, closely followed by the CC k-nearest anytime RRT, whose estimated mean is only 0.090% to 3.076% larger than the CC informed RRT*'s mean. The CC basic anytime RRT comes last, lagging by a significant margin. Its mean is up to 16.904% larger than that of the CC k-nearest anytime RRT and up to 19.704% larger than that of the CC informed RRT*.

Apart from comparing the mean path cost per iteration between the benchmark path-planning algorithms, it is also interesting to compare the mean path cost of each algorithm to its minimum. For this purpose, the minimum and maximum costs have been plotted along with the estimated mean and its confidence intervals in Figure 5.12. From these plots, it is clear that the CC informed RRT*'s mean stays closest to its minimum. Numerically, its mean is only up to 9.3798% larger than its minimum. The CC k-nearest anytime RRT follows closely, with its mean being up to 10.775% larger than its minimum. The CC anytime RRT again comes last, with its minimum being up to 14.706% larger than its minimum.

Finally, in terms of variability of path costs returned by the three benchmark path-planning algorithms, the costs of solutions returned by the CC basic anytime RRT are the most variable. As evident from the interquartile range (IQR) and standard deviation in Table 5.1, its IQR and standard deviation are, frequently, at least two times those of the other two algorithms. The IQR and standard deviation of the CC k-nearest anytime RRT are close to that of informed RRT*.

The foregoing analyses and comparisons have established a baseline for the head-to-head performance of the developed benchmark path-planning algorithms. The analyses confirm our expected head-to-head performance of the algorithms, with the CC informed RRT* leading in all considered criteria. It is closely followed by the CC k-nearest anytime RRT. The CC basic anytime RRT lags behind by a large margin in all aspects.

With this baseline in place, we now proceed to investigate the effect of incorporating a path-optimisation step to each of the benchmark path-planning algorithms as a way of accelerating their rate of convergence. The main focus is on investigating how such a path-optimisation step can be of value to a quick, almost-surely suboptimal path planners, like the CC basic anytime RRT, in making them attain performance that is comparable or better than that of asymptotically optimal path planners, such as the CC informed RRT*. For this purpose, once the path-optimisation step is incorporated to each path planner, a similar performance comparison will be done for each improved algorithm. Then, by using the results presented in this section as a baseline, a relative improvement for each algorithm from its baseline performance can be computed. Also, the

performance of each improved algorithm can be compared with the baseline performance of any of the other algorithms. This is the subject of the next subsection.

5.6.3 Analyses and Comparison of the Optimised Global Path Planners

The preceding subsection analysed the developed benchmark path planners and established a baseline for their head-to-head performance. This subsection presents analyses of the optimised path-planning algorithms, i.e. the benchmark path planners augmented with a path-optimisation step. The path-optimisation step takes the form of the various path-optimisation algorithms developed in Section 5.5. These are path pruning, random shortcut, the wrapping process and gradient-based path optimisation. As a result, each optimised path planner has four versions, each based on one of the path optimisers. The optimised path planners resulting from the application of a given path-optimisation algorithm to accelerate the rate of convergence of the three benchmark path planner are presented together, with the aim being to study the effectiveness of that path-optimisation algorithm in accelerating the rate of convergence of each of the benchmark path planners. Subsections 5.6.3.1, 5.6.3.2, 5.6.3.3, and 5.6.3.4 respectively present versions of the optimised path planners that are realised through the use of path pruning, random shortcut, the wrapping process, and gradient-based path optimisation. The analyses of each set of optimised path planners follow a format similar to that used in the analyses of the benchmark path planners presented in the previous subsection.

5.6.3.1 Optimised Path Planners based on Path Pruning

This subsection presents analyses of our first set of optimised path planners. These optimised path planners have been formed by augmenting the benchmark path planners with a path-pruning step, whose role is to reduce the length of each newly-found solution before the cost of this new solution is used to bound the search space in the subsequent search.

Similarly to the analyses of the benchmark paths planners, the analyses presented here are based on results obtained from 400 runs of each of the optimised path planners. The results are again presented graphically through a percentile plot, shown in Figure 5.13, with a numerical summary of some statistics provided in Table 5.2. Figure 5.14 plots the estimated mean cost, with 95% confidence bounds. The format of the presented results is the same as that of the results of the benchmark global planners. Again, the aim is to ensure ease of comparison between the optimised and baseline versions of the path planners.

Starting with the minimum costs of solutions returned by the algorithms, it is interesting to note that the optimised CC basic anytime RRT is able to find paths that are shorter than those found by than the optimised CC k-nearest anytime RRT and the optimised CC informed RRT*, as clearly shown by the numerical summary of statistics given in Table 5.2. This is in contrast to the performance of its benchmark path planner, the CC basic anytime RRT, which came last among the benchmark path planners in this regard. Relative to its baseline minimum, the optimised CC basic RRT's minimum is at up to 10.26% smaller than the baseline. With respect to the other benchmark path planners, the minimum of the optimised CC basic anytime RRT is up to 2.69% smaller than the CC k-nearest anytime RRT's minimum, and up to 2.20% smaller than the CC informed RRT*'s minimum. The optimised CC informed RRT* comes second with regard to minimum costs among the optimised path planners based on path pruning; its minimum is up to 2.34% larger than the optimised CC basic anytime RRT's minimum. The optimised CC k-nearest anytime RRT then comes last, with its minimum being upto 2.98% larger than the optimised CC basic anytime RRT's minimum. Relative to their baselines, the minimum costs of the optimised CC k-nearest anytime RRT and the optimised CC informed RRT* are respectively up to 1.98% and 2.42% smaller. This means that the path pruning method is more effective in optimising paths produced by the CC basic anytime RRT than it is in optimising paths produced by the CC k-nearest anytime RRT and the CC informed RRT*. This is to be expected since path pruning depends on the visibility between non-consecutive path nodes and paths produced by the CC basic anytime RRT typically have more path nodes than paths produced by the two other benchmark path planners. This makes it more likely for the path pruner to find shortcuts in paths produced by the CC basic anytime RRT than in paths produced by the other two benchmark path planners.

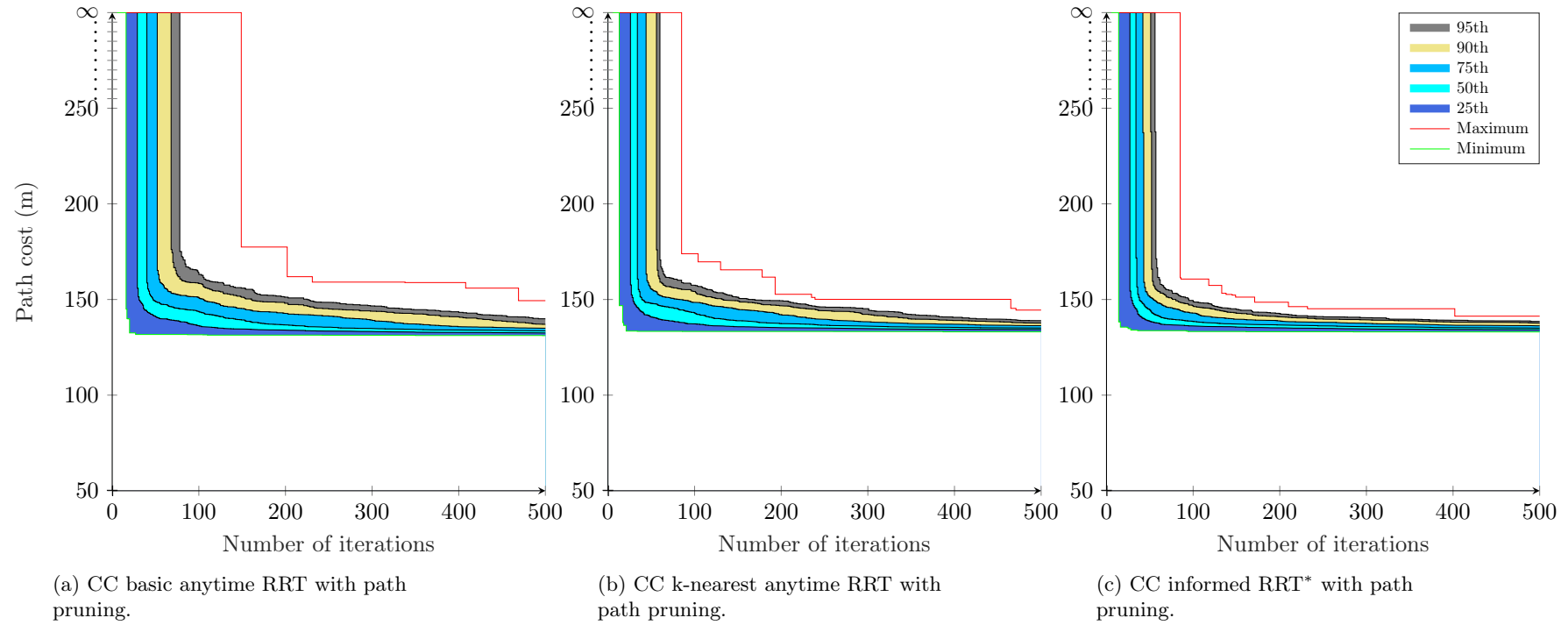


Figure 5.13: A percentile plot showing the evolution or improvement of path cost over an increasing number of iterations for 400 runs of each of the three optimised global path-planning algorithms, when path pruning is applied to accelerate their rate of convergence.

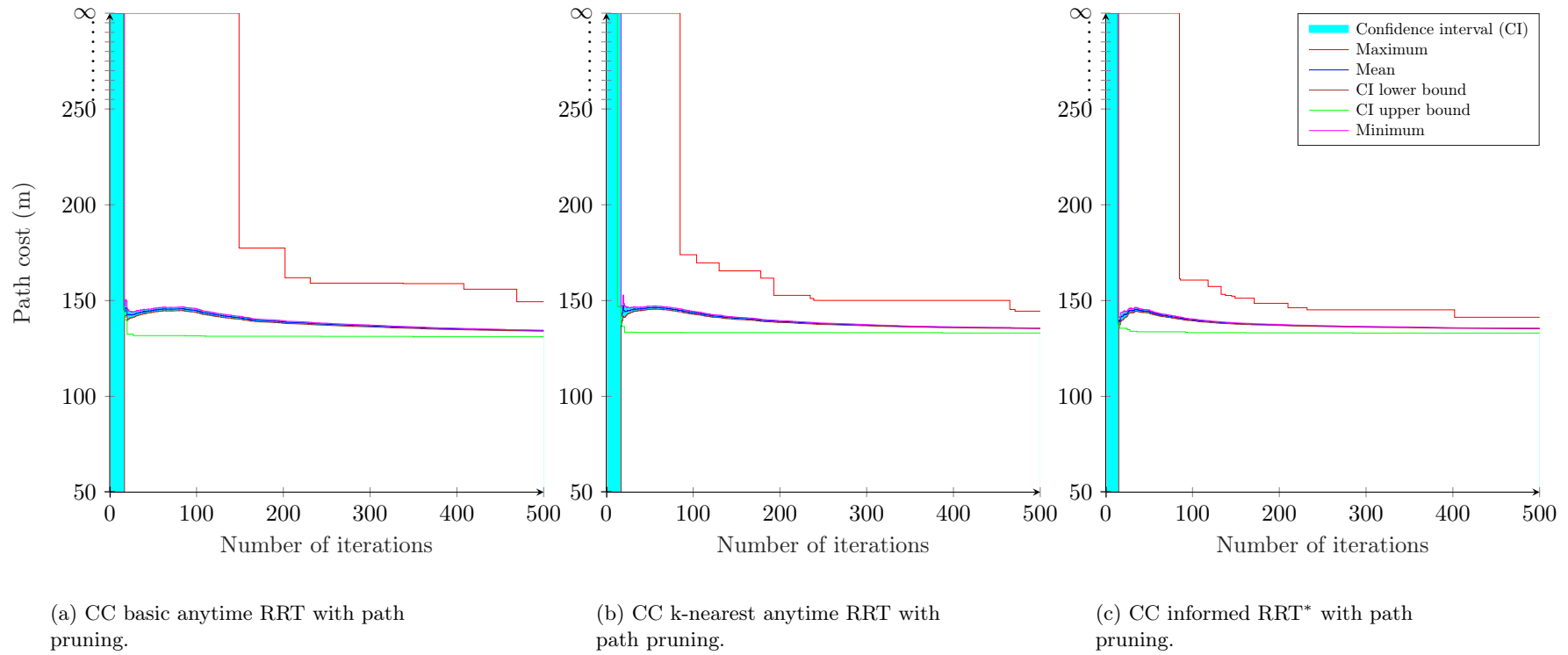


Figure 5.14: A plot of the mean path cost per iteration, with a confidence interval, for 400 independent runs of each optimised global path-planning algorithm based on path pruning.

	Iteration No.	Min.	Max.	q_1	q_2	q_3	IQR	90 th	95 th	Mean	Std. dev.
optimised CC basic anytime RRT	50	131.700	∞	140.971	148.873	∞	∞	∞	∞	144.980	8.272
	100	131.598	∞	136.233	143.579	150.793	14.560	158.350	163.096	144.654	9.620
	150	131.363	177.430	134.578	139.305	145.272	10.694	151.171	155.952	141.000	7.781
	200	131.363	177.430	133.959	136.685	142.708	8.749	148.035	151.220	138.778	6.431
	250	131.299	159.093	133.477	135.526	141.292	7.815	145.017	148.507	137.614	5.374
	300	131.299	159.093	133.249	134.839	139.062	5.813	143.853	146.640	136.722	4.870
	350	131.176	158.854	132.893	134.379	137.196	4.303	142.436	144.680	135.890	4.270
	400	131.176	158.854	132.788	134.037	135.769	2.981	140.876	143.281	135.160	3.711
	450	131.176	155.940	132.626	133.882	135.311	2.685	138.655	141.531	134.662	3.147
	500	131.106	149.396	132.519	133.640	134.976	2.457	137.034	139.930	134.272	2.633
optimised CC k-nearest anytime RRT	50	133.345	∞	140.868	147.952	155.394	14.526	∞	∞	146.214	7.773
	100	133.288	173.931	137.276	142.901	148.469	11.193	154.198	157.447	143.639	7.672
	150	133.288	165.600	135.756	138.888	144.845	9.089	149.190	151.112	140.753	5.902
	200	133.288	152.722	135.246	137.364	142.026	6.780	146.676	149.362	139.01	4.800
	250	133.288	150.081	135.065	136.556	139.653	4.588	144.127	146.109	137.894	3.905
	300	133.260	150.081	134.799	136.085	138.362	3.563	142.235	145.195	137.181	3.387
	350	133.260	150.081	134.475	135.588	137.424	2.949	139.760	142.106	136.395	2.672
	400	133.081	150.081	134.427	135.423	137.060	2.633	139.017	140.676	136.047	2.316
	450	133.081	150.081	134.340	135.176	136.531	2.191	138.417	139.734	135.786	2.164
	500	133.081	144.466	134.286	135.077	136.343	2.057	137.666	138.938	135.520	1.803
optimised CC informed RRT*	50	133.69	∞	138.569	144.388	151.252	12.683	∞	∞	144.177	7.463
	100	133.257	160.689	136.158	138.508	142.888	6.730	146.397	148.882	139.911	4.896
	150	133.257	151.242	135.567	137.138	139.688	4.121	142.787	144.806	137.993	3.313
	200	133.150	148.574	135.194	136.672	138.786	3.592	140.908	142.554	137.259	2.716
	250	133.150	145.170	134.894	136.184	138.065	3.171	139.750	140.964	136.685	2.296
	300	133.049	145.170	134.590	135.790	137.527	2.937	139.376	140.496	136.282	2.173
	350	133.049	145.170	134.492	135.619	137.154	2.662	138.875	139.688	136.037	2.021
	400	133.049	145.170	134.320	135.274	136.506	2.186	138.121	139.143	135.655	1.721
	450	133.049	141.284	134.267	135.171	136.424	2.157	138.045	138.831	135.536	1.598
	500	133.049	141.284	134.190	135.024	136.252	2.062	137.627	138.537	135.406	1.534

Table 5.2: Numerical summary of the statistics of the optimised path planners based on path pruning.

In terms of the maximum cost of solutions returned by the three optimised path planners, the trend is the same as that observed among the benchmark path planners. The optimised CC informed RRT* returns maximum-cost paths with lower costs compared to both the optimised CC basic RRT and optimised CC k-nearest anytime RRT. The optimised CC basic anytime RRT's maximum cost is up to 19.42% larger than the optimised CC informed RRT*'s maximum cost and the optimised CC k-nearest anytime RRT's maximum cost is up to 11.40% larger than that of the CC informed RRT*. While this trend remains the same as in the benchmark path planners, a significant improvement can be observed for the optimised CC basic anytime RRT from its baseline. Particularly, the optimised CC basic anytime RRT's maximum is upto 27.33% smaller than its baseline's maximum. On the other hand the optimised CC k-nearest anytime RRT and optimised CC informed RRT* have only improved by upto 7.767% and 6.70%, respectively from their baseline maxima.

The trend observed on all the percentiles of the optimised path-planning algorithms is completely different from that observed in the benchmark path planners. In the analyses of the benchmark path planners, the costs of the 75th percentile of the CC k-nearest anytime RRT and the CC informed RRT* were found to be below the CC basic anytime RRT's 25th percentile. Moreover, for iterations beyond 350, it was observed that the CC informed RRT*'s maximum was below the CC basic anytime RRT's 25th percentile. This proved how highly suboptimal the solutions found by the CC basic anytime RRT are. With path pruning used to accelerate the rate of convergence of the CC basic anytime RRT, this is no longer the case. This can be seen by comparing Figures 5.11 and 5.13. as well as Tables 5.1 and 5.2. Interestingly, as can be seen in Table 5.2, the optimised CC basic anytime RRT's 25th to 75th percentiles frequently have lower costs than corresponding percentiles of the optimised CC informed RRT*. Percentiles of the optimised CC basic anytime RRT that consistently have higher costs compared to their optimised CC informed RRT* counterparts are then the 90th and 95th. Importantly, these do not differ by large values from their optimised CC informed RRT* counterparts.

As shown in Figure 5.13 and Table 5.2, the mean path costs of the optimised path-planning algorithms are all in the same range, unlike in the benchmark path planners where the CC basic anytime RRT's mean was significantly above the mean path costs of the other two benchmark path planners. It is interesting to note that above 350 iterations, the mean path cost of the optimised CC basic anytime RRT is consistently below the mean path costs of the other two optimised path planners.

Finally, the IQR and standard deviation of the optimised path planners are in the same range. This is in contrast to the benchmark path-planning algorithms, where the CC basic anytime RRT's IQR and standard deviation were, frequently, at least two times larger than those of the other two benchmark path planners.

Overall, the analyses of optimised path planners based on path pruning that have been performed in this subsection indicate that a significant improvement to the CC basic anytime RRT is realised by incorporating the path-pruning step, while not as much improvement is achieved for the CC k-nearest anytime RRT and the CC informed RRT*. With this improvement, the resulting optimised path-planning algorithms have comparable performance, as opposed to the benchmark path-planning algorithms where the CC basic anytime RRT lagged by a significant margin in all aspects. We now proceed to investigate the effectiveness of random shortcut in accelerating the rate of convergence of the benchmark path planners.

5.6.3.2 Optimised Path Planners based on Random Shortcut

The preceding subsection analysed our first set of optimised path-planning algorithms. These algorithms were formed by using path pruning to accelerate the rate of convergence of the benchmark path planners, namely the CC basic anytime RRT, the CC k-nearest anytime RRT and the CC informed RRT*. Path pruning was found to be more effective in accelerating convergence of the CC basic anytime RRT compared to the other two benchmark path planners. In this section we present analyses for our second set of optimised path planners. These are obtained by applying our second path-optimisation technique, random shortcut, to accelerate the rate of convergence of the benchmark path planners. For the analyses, we again make use of a percentile plot (shown in

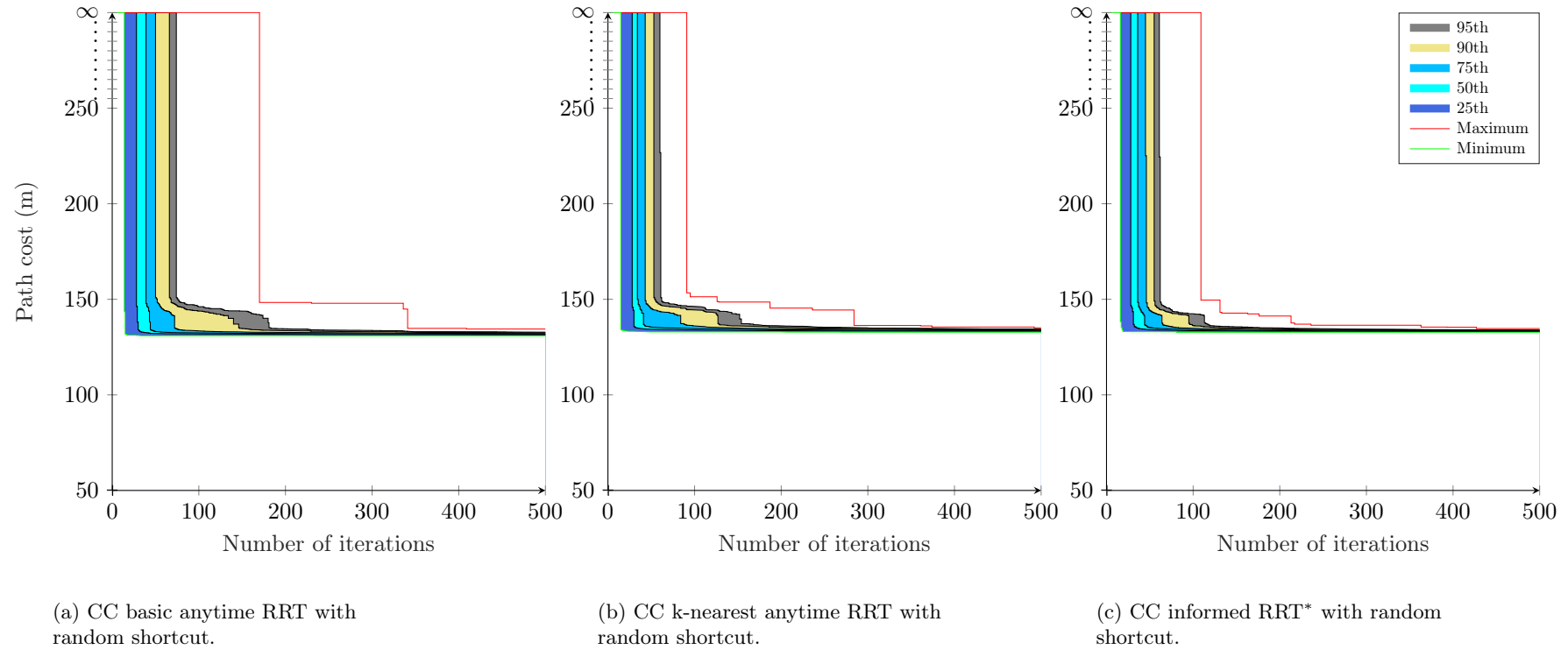


Figure 5.15: A percentile plot showing the evolution or improvement of path cost over an increasing number of iterations for 400 runs of each of the three optimised global path-planning algorithms, when random shortcut is applied to accelerate their rate of convergence.

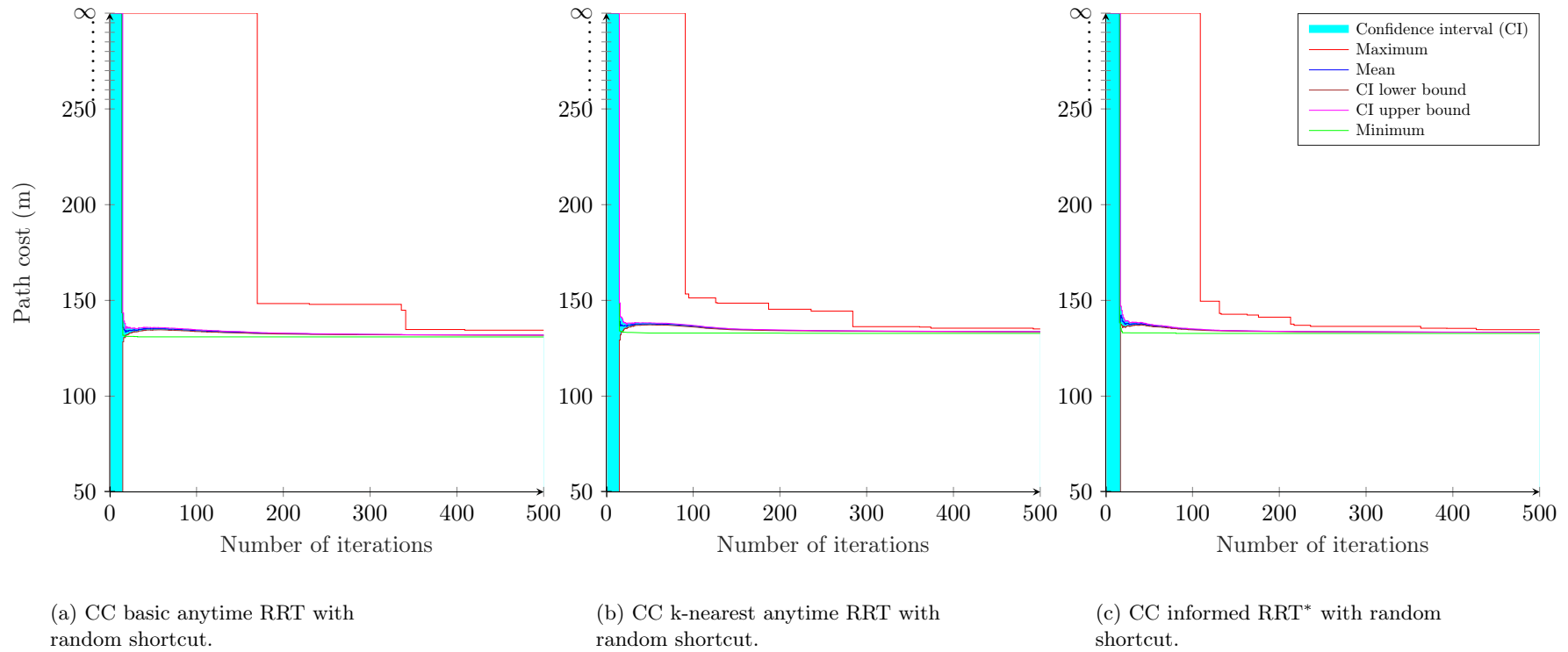


Figure 5.16: A plot of the mean path cost per iteration, with a confidence interval, for 400 independent runs of each optimised global path-planning algorithm based on the random shortcut.

	Iteration No.	Min.	Max.	q_1	q_2	q_3	IQR	90 th	95 th	Mean	Std. dev.
optimised CC basic anytime RRT	50	130.956	∞	132.080	133.100	150.906	18.826	∞	∞	135.206	5.634
	100	130.956	∞	131.853	132.393	133.430	1.577	143.831	146.117	134.186	4.606
	150	130.956	∞	131.779	132.188	132.880	1.101	134.663	143.803	133.329	3.584
	200	130.956	148.351	131.699	132.066	132.629	0.930	133.575	134.589	132.655	2.460
	250	130.911	147.926	131.609	131.978	132.512	0.903	133.201	133.881	132.336	1.824
	300	130.911	147.926	131.566	131.913	132.383	0.817	133.021	133.656	132.156	1.385
	350	130.911	134.788	131.534	131.861	132.222	0.685	132.696	133.050	131.962	0.609
	400	130.911	134.788	131.500	131.823	132.192	0.692	132.688	133.035	131.928	0.596
	450	130.911	134.444	131.474	131.793	132.149	0.676	132.592	132.955	131.884	0.566
	500	130.911	134.444	131.451	131.756	132.082	0.631	132.525	132.811	131.839	0.532
optimised CC k-nearest anytime RRT	50	132.940	∞	134.315	135.446	145.144	10.829	∞	∞	137.790	5.076
	100	132.940	151.326	133.933	134.628	136.024	2.091	144.461	146.310	136.494	4.324
	150	132.940	148.567	133.693	134.258	134.970	1.278	136.078	142.154	134.835	2.346
	200	132.892	145.388	133.570	134.042	134.641	1.072	135.372	136.076	134.313	1.398
	250	132.843	144.403	133.475	133.916	134.404	0.928	134.946	135.402	134.065	0.954
	300	132.784	136.295	133.433	133.817	134.262	0.829	134.754	134.980	133.898	0.610
	350	132.784	136.295	133.395	133.717	134.149	0.755	134.564	134.829	33.809	0.559
	400	132.784	135.491	133.324	133.656	134.053	0.728	134.361	134.587	133.713	0.481
	450	132.784	135.491	133.317	133.587	133.956	0.639	134.287	134.479	133.664	0.458
	500	132.784	135.092	133.310	133.551	133.890	0.579	134.195	134.435	133.632	0.428
optimised CC informed RRT*	50	133.041	∞	134.087	135.205	144.026	9.939	∞	∞	136.961	4.404
	100	132.762	∞	133.582	134.100	134.827	1.245	136.233	142.388	134.857	2.513
	150	132.762	142.773	133.443	133.759	134.413	0.970	134.995	135.601	134.072	1.174
	200	132.762	141.247	133.327	133.607	134.093	0.767	134.599	135.037	133.817	0.796
	250	132.762	136.434	133.259	133.530	133.857	0.598	134.326	134.595	133.624	0.516
	300	132.762	136.434	133.219	133.437	133.753	0.533	134.102	134.349	133.536	0.457
	350	132.762	136.434	133.188	133.375	133.661	0.474	134.029	134.224	133.473	0.426
	400	132.762	135.394	133.157	133.337	133.563	0.406	133.950	134.105	133.410	0.370
	450	132.762	134.700	133.142	133.326	133.537	0.395	133.837	134.040	133.372	0.328
	500	132.762	134.700	133.129	133.298	133.510	0.381	133.775	133.971	133.343	0.312

Table 5.3: Numerical summary of the statistics of the optimised path planners based on random shortcut at selected number of iterations.

Figure 5.15), a summary of path cost statistics (given in Table 5.3), and a plot of the mean path cost, estimated with 95% confidence (shown in Figure 5.16).

By visual inspection of Figure 5.15, clear convergence can be observed for all the optimised path-planning algorithms as the maximum and all the percentiles decrease progressively, tending towards the minimum as the number of iterations increases.

Starting with the minimum costs of solutions returned by the algorithms, the optimised CC basic anytime RRT is able to find paths that are shorter than paths found by the optimised CC k-nearest anytime RRT and the optimised CC informed RRT*. This can be clearly seen from Table 5.15. The optimised CC informed RRT* comes second in terms of the minimum path cost, and the optimised CC k-nearest anytime RRT comes last. In comparing the minima of the optimised path planners to their baseline minima, we find that the optimised CC basic anytime RRT has a minimum that is up to 11.17% smaller than its baseline while the optimised CC k-nearest anytime RRT's minimum is only up to 5.23% smaller than its baseline, and lastly, the optimised CC informed RRT*'s minimum is only up to 4.30% smaller than its baseline.

We now analyse the optimised path-planning algorithms according to the maximum costs of solutions they return. It is interesting to note that while the optimised CC informed RRT* leads the optimised path planners in this criterion for iterations below 350, above that number of iterations, the optimised CC basic anytime RRT leads. In both cases, the optimised k-nearest anytime RRT comes second. When compared to the minimum of its benchmark, the optimised CC basic anytime RRT's maximum is up to 34.50% smaller than the benchmark. On the other hand, the maxima of the optimised CC k-nearest anytime RRT and optimised CC informed RRT* are respectively only up to 14.04% and 12.13% smaller than their baselines.

In terms of the percentiles, as evident from Table 5.15, the optimised CC basic anytime RRT's 25th and 50th percentiles are consistently below the same percentiles of the other two optimised path-planning algorithms. Also, from 100 iterations and above, the optimised CC basic anytime RRT's 75th percentile is consistently below the same percentiles of the other two optimised path planners. From 200 iterations onwards, the optimised CC basic anytime RRT's 90th and 95th percentiles are consistently below corresponding percentiles of the other two optimised path planners. Finally, from 350 iterations onwards, the optimised CC basic anytime RRT's 90th and 95th percentiles are consistently below both the optimised CC informed RRT*'s 25th percentile and the CC k-nearest anytime RRT's minimum.

The mean path cost of the optimised CC basic anytime RRT is consistently below the mean path costs of both the optimised CC k-nearest anytime RRT and optimised CC informed RRT*. Interesting to note is the fact that from 200 iterations onwards, the optimised basic anytime RRT's mean path cost is consistently below the minimum path costs of the other two optimised path planners. When compared to the baseline mean path costs, the optimised CC basic anytime RRT's mean path cost is up to 21.29% smaller than its baseline while the mean path costs of the optimised CC k-nearest anytime RRT and the optimised CC informed RRT* are respectively only up to 7.24% and 6.05% smaller than their baselines.

In terms of path cost variability, the IQR and standard deviation can be seen converging towards zero for all the optimised path planners as the number of iterations increases.

From the above analyses, it is clear that the contention for the best among the three optimised path planners based on random shortcut is between the optimised CC basic anytime RRT and the optimised CC informed RRT*. The optimised CC basic anytime RRT, without doubt, does well with regard to the minimum, the mean path costs, as well as the 25th and 50th percentiles. With regard to the rest of the considered criteria, namely the maximum cost as well as the 75th, 90th and 95th percentiles, adjudication is required. As stated in the analyses, the optimised CC informed RRT* does well in producing lower maximum costs than the optimised CC basic anytime RRT in early iterations, while later on, the optimised CC basic anytime RRT has lower maximum costs. Also, with regard to the 75th, 90th and 95th percentiles, early on, the optimised CC informed RRT leads, while the optimised CC basic anytime RRT takes the lead later on, with the 90th and 95th percentiles being consistently below the optimised CC informed RRT*'s 25th percentile. To decide on these cases, it is necessary to recall the computational cost of an iteration of each of the benchmark path-planning algorithms. An iteration of the CC informed RRT* is generally more expensive compared to an iteration of the CC basic anytime RRT due to the consideration of multiple neighbours for tree extension and the presence of tree rewiring process in the CC informed

RRT*. With this consideration in mind, the optimised CC basic anytime RRT may not necessarily be set back by that fact that it only starts to find lower maximum costs and low-cost 75th, 90th and 95th percentiles in later iterations.

Overall, the application of random shortcut to accelerate the convergence of the benchmark path planners leads to clear convergence towards the minima, as seen in the percentile plots. The achieved improvement is significantly greater than that achieved by applying path pruning; even noticeable by inspection of the percentile and mean path cost plots. Among the optimised path planners, the optimised CC basic nearest anytime RRT experiences significant improvement compared to the other two optimised path planners as pointed out in the preceding analyses. Moreover, it is important to point out that from the experiments performed, the application of random shortcut in the optimised CC informed RRT causes the algorithm to be slow, yet for the optimised CC basic anytime RRT the algorithm's speed remains acceptable. We now proceed to investigate the effectiveness of the wrapping process in accelerating the convergence of the benchmark path planners.

5.6.3.3 Optimised Path Planners based on the Wrapping Process

We now present the analyses of our third set of path-optimisation algorithms. These algorithms are realised by applying the wrapping process to accelerate the convergence of the benchmark path planners, namely the CC basic anytime RRT, the CC k-nearest anytime RRT and the CC informed RRT*. We again base our analyses on a percentile plot, a summary of statistics and a plot of the mean path cost. These are given in Figure 5.17, Table 5.4 and Figure 5.18.

Just like in the case of the optimised path planners based on random shortcut of the previous subsection, clear convergence can be observed as the maximum and all percentiles of all of the optimised path planner decay, tending towards the minimum as iterations increase.

We begin by considering the minimum costs of solutions returned by the optimised path planners. Like in the cases of path-pruning and random-shortcut steps, the optimised CC basic anytime RRT is able to find paths that are shorter than paths found by both the optimised CC k-nearest anytime RRT and the optimised CC informed RRT*. This shown by the minimum of the optimised CC basic anytime RRT, which is consistently below the minima of the two other optimised path-planning algorithms. The optimised CC informed RRT* comes second in this regard, with its minimum cost consistently below that of the optimised CC k-nearest anytime RRT. When comparing the minimum of the optimised CC basic anytime RRT to its baseline, we find that it is up to 11.40% smaller than the baseline. On the other hand the minima of the optimised CC k-nearest anytime RRT and the optimised CC informed RRT* are respectively up to 5.56% and 7.81% smaller than their baselines.

In terms of the maximum path cost returned by each of the optimised path-planning algorithms, the optimised CC informed RRT* returns low-cost maximum costs compared to the other two optimised path planners for iterations less than 450. Interestingly from 450 iterations upwards, the optimised CC basic anytime RRT's maximum is consistently below the maxima of the two other optimised path planners. In comparison to its baseline, the maximum of the optimised CC basic anytime RRT is up to 34.03% smaller. On the other hand the maxima of the optimised CC k-nearest anytime RRT and the optimised CC informed RRT* are respectively up to 14.16% and 14.81% smaller than their baselines.

Considering the percentiles, we notice that the optimised CC basic anytime RRT's 25th and 50th percentiles are consistently below corresponding percentiles in the other two optimised path planners. We also note that from 150 iterations onwards, the optimised CC basic anytime RRT's 75th percentile is also consistently below the corresponding percentile in the other two optimised path-planning algorithms. More interestingly, from 200 iterations onwards, the optimised CC basic anytime RRT's 90th percentile is consistently less than the minima of the other two optimised path-planning algorithms, and moreover, from 350 iterations and beyond, the optimised CC basic anytime RRT's 95th percentile consistently stays below both the optimised CC informed RRT*'s 25th percentile and from 450 iterations onwards, it gets below the optimised CC informed RRT*'s minimum.

In terms of the mean path costs of the three optimised path-planning algorithms, the CC basic anytime RRT's mean path cost is consistently below the mean path costs of the other two

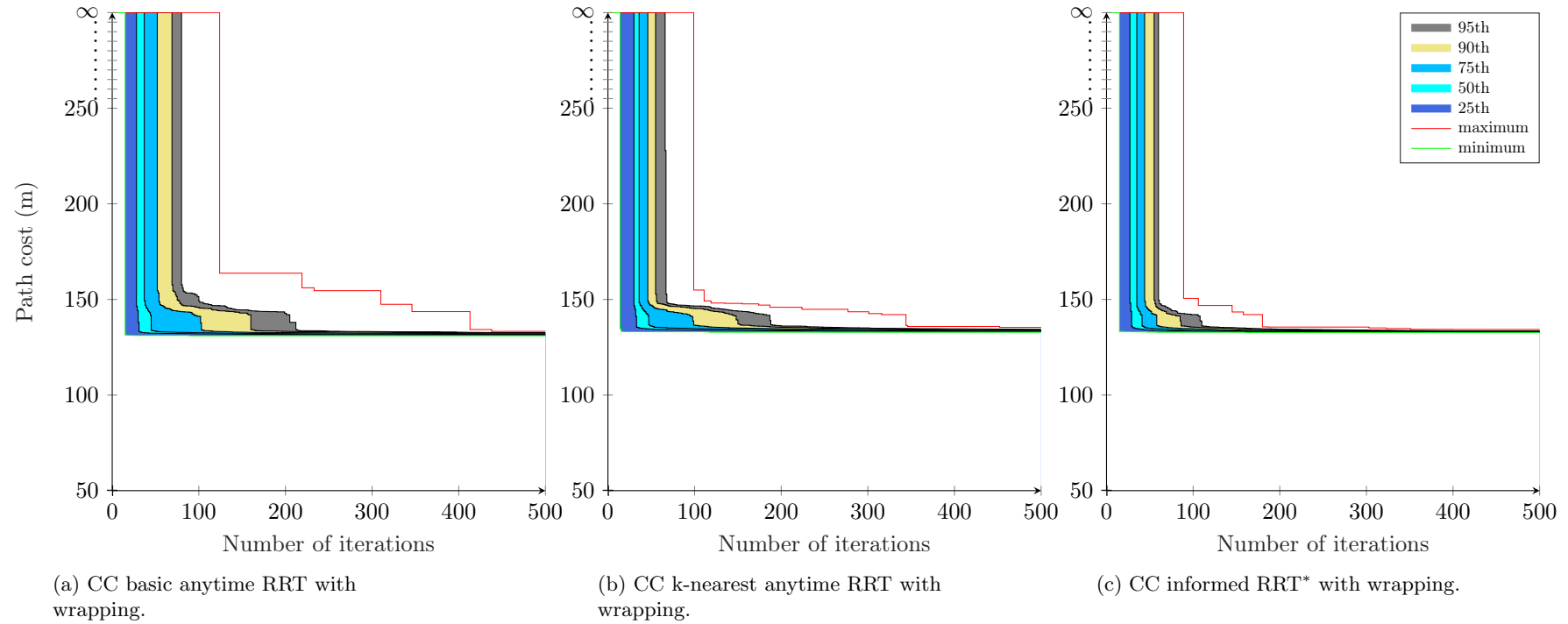


Figure 5.17: A percentile plot showing the evolution or improvement of path cost over an increasing number of iterations for 400 runs of each of the three optimised path planners that have been formed by incorporating the wrapping process into each of the benchmark path planners.

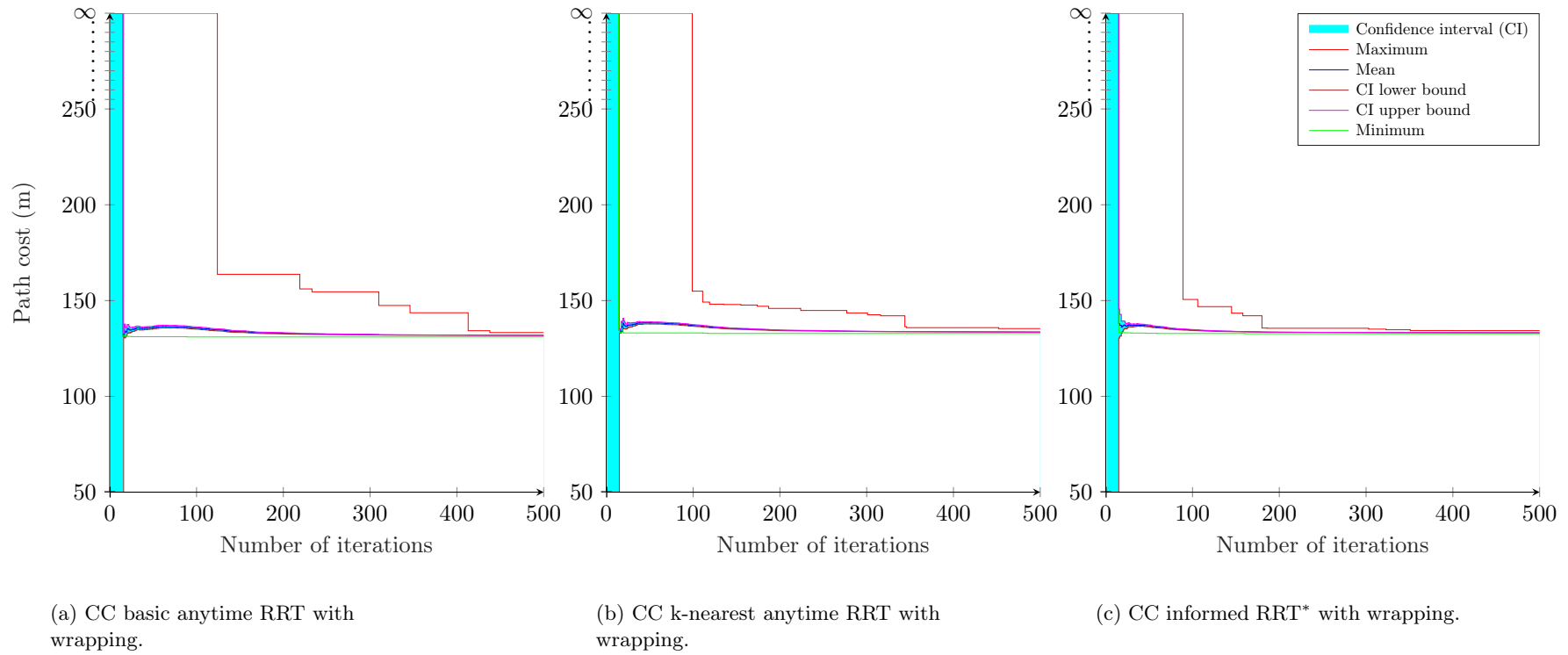


Figure 5.18: A plot of the mean path cost per iteration, with a confidence interval, for 400 independent runs of each optimised global path-planning algorithm based on the wrapping process.

	Iteration No.	Min.	Max.	25 th	50 th	75 th	IQR	90 th	95 th	Mean	Std. dev.
optimised CC basic anytime RRT	50	131.146	∞	132.310	133.354	∞	∞	∞	∞	135.941	6.319
	100	131.012	∞	132.002	132.585	141.190	9.188	145.63	149.169	135.660	6.250
	150	131.012	163.717	131.802	132.291	132.865	1.063	142.870	144.549	133.787	4.610
	200	131.012	163.717	131.700	132.123	132.645	0.945	133.276	142.769	132.922	3.396
	250	131.012	154.575	131.636	131.994	132.547	0.911	133.076	133.362	132.404	2.169
	300	131.012	154.575	131.579	131.903	132.330	0.751	132.865	133.202	132.221	1.914
	350	131.012	143.640	131.542	131.807	132.261	0.719	132.660	133.076	131.981	0.924
	400	131.012	143.640	131.527	131.767	132.169	0.642	132.596	132.852	131.909	0.770
	450	131.012	133.294	131.504	131.736	132.074	0.570	132.448	132.670	131.827	0.449
	500	131.012	133.294	131.493	131.717	132.039	0.546	132.396	132.654	131.803	0.433
optimised CC k-nearest anytime RRT	50	133.121	∞	134.512	136.081	146.935	12.423	∞	∞	138.397	5.310
	100	133.121	154.919	134.050	134.754	136.508	2.458	145.293	146.556	137.002	4.603
	150	132.857	147.946	133.794	134.371	135.050	1.256	137.165	143.858	135.306	3.013
	200	132.857	145.907	133.615	134.071	134.658	1.043	135.438	136.101	134.448	1.754
	250	132.823	144.824	133.479	133.828	134.446	0.967	134.996	135.477	134.164	1.442
	300	132.823	143.549	133.401	133.727	134.212	0.811	134.655	135.119	133.918	0.974
	350	132.823	135.858	133.327	133.609	133.995	0.668	134.472	134.748	133.719	0.531
	400	132.823	135.858	133.282	133.563	133.885	0.603	134.352	134.616	133.653	0.492
	450	132.823	135.858	133.258	133.492	133.817	0.559	134.210	134.517	133.590	0.460
	500	132.823	135.384	133.235	133.448	133.777	0.542	134.100	134.356	133.541	0.424
optimised CC informed RRT*	50	132.929	∞	133.994	134.932	144.014	10.020	∞	∞	136.827	4.543
	100	132.837	150.575	133.410	133.837	134.694	1.284	135.509	142.077	134.608	2.551
	150	132.837	143.440	133.256	133.548	134.008	0.752	134.710	135.132	133.804	1.115
	200	132.713	135.553	133.177	133.351	133.690	0.513	134.147	134.422	133.486	0.461
	250	132.713	135.553	133.139	133.279	133.535	0.396	133.890	134.197	133.389	0.408
	300	132.713	135.553	133.102	133.222	133.411	0.309	133.746	133.965	133.303	0.329
	350	132.713	134.770	133.081	133.181	133.355	0.274	133.571	133.762	133.245	0.264
	400	132.713	134.312	133.056	133.156	133.293	0.237	133.480	133.588	133.201	0.224
	450	132.712	134.306	133.049	133.136	133.265	0.216	133.414	133.550	133.176	0.213
	500	132.712	134.306	133.043	133.121	133.222	0.179	133.371	133.494	133.154	0.202

Table 5.4: Numerical summary of the statistics of optimised path planners based on the wrapping process, at selected number of iterations.

optimised path planners. Moreover, from 250 iterations onwards, the optimised CC basic anytime RRT's mean is consistently below the minima of the other two optimised path-planning algorithms. In comparison to its baseline, the mean path cost of the optimised CC basic anytime RRT is up to 20.45% smaller. On the other hand, the optimised CC k-nearest anytime RRT and the optimised CC informed RRT* are respectively up to 7.99% and 7.07% smaller than their baselines. Like in the optimised path planners based on random shortcut, the IQR and standard deviation of the optimised path planners can be seen converging towards zero for all the optimised path planners as the number of iterations increases.

The outcome of the analyses of the wrapping-based optimised path planners is similar to that presented in the previous subsection for optimised path planners based on random shortcut. From the above analyses it is clear that the contention for the best among the optimised path planners is again between the optimised CC basic anytime RRT and the optimised CC informed RRT*. The optimised CC basic anytime RRT again, without doubt, does well with regard to the minimum, the mean path costs, as well as the 25th and 50th percentiles. As pointed out in the analyses, early on, the optimised informed RRT*'s 75th, 90th and 95th percentiles as well as its maximum is consistently below the corresponding percentiles and maximum of the optimised CC basic anytime RRT. However, later on, the optimised CC basic anytime RRT takes the lead with regard to these statistics. Again due to the differences in the computational cost of an iteration of the corresponding benchmark algorithms (CC basic anytime RRT and CC informed RRT*), the optimised CC basic anytime RRT may not necessarily be set back by the fact that it only produces low values for these statistics in later iterations, since it is generally faster than the optimised CC informed RRT*.

Overall, incorporating the wrapping process to accelerate the convergence of the benchmark path planners leads to clear convergence towards the minima, as seen in the percentile plots. The achieved improvement is similar to that observed in the application of random shortcut, with the optimised CC basic anytime RRT experiencing significant improvement compared to the other two optimised path planners. The difference is that with the wrapping process, the optimised CC informed RRT* was not as slow as in the case of random shortcut. We now proceed to the analyses of our last set of optimised path planners, which are formed by incorporation a gradient-based path-optimisation step into the benchmark path planners.

5.6.3.4 Optimised Path Planners based on Gradient-based Path Optimisation

In this subsection, we present analyses for our last set of optimised path-planning algorithms. These algorithms are formed by incorporating gradient-based path optimisation to each of the benchmark path planners. The performance of the optimised path planners is compared with their baseline performance to ascertain the effectiveness of the gradient-based path-optimisation step in accelerating the convergence of the benchmark path planners.

Figure 5.19, Table 5.5 and Figure 5.20 respectively give percentile plots, a numeric summary of statistics and plots of the mean path costs for the three optimised path-planning algorithms.

Like in the case of the optimised path planners that are based on path pruning, random shortcut and the wrapping process, the optimised CC basic anytime RRT is able to find paths that are shorter than paths found by the optimised CC k-nearest anytime RRT and the optimised CC informed RRT*. This is evident in Table 5.5, where the optimised CC basic anytime RRT's minimum is consistently below the minima of the two other optimised path-planning algorithms. The optimised CC informed RRT* comes second in this regard, with its minimum consistently below the optimised CC k-nearest anytime RRT's minimum. In comparison to its baseline, the minimum of the optimised CC basic anytime RRT is 10.89% smaller. On the other hand, the minima of the optimised CC k-nearest anytime RRT and the optimised CC informed RRT* are respectively 4.81% and 7.35% smaller than their baselines.

In terms of the maximum cost, the same trend observed among the benchmark path planners is observed here. The optimised CC informed RRT* returns lower maximum costs compared to the optimised CC basic anytime RRT and the optimised CC k-nearest anytime RRT. It is followed by the optimised CC basic anytime RRT, and last comes the optimised CC k-nearest anytime RRT. The optimised CC basic anytime RRT's maximum is up to 33.41% smaller than its baseline. On

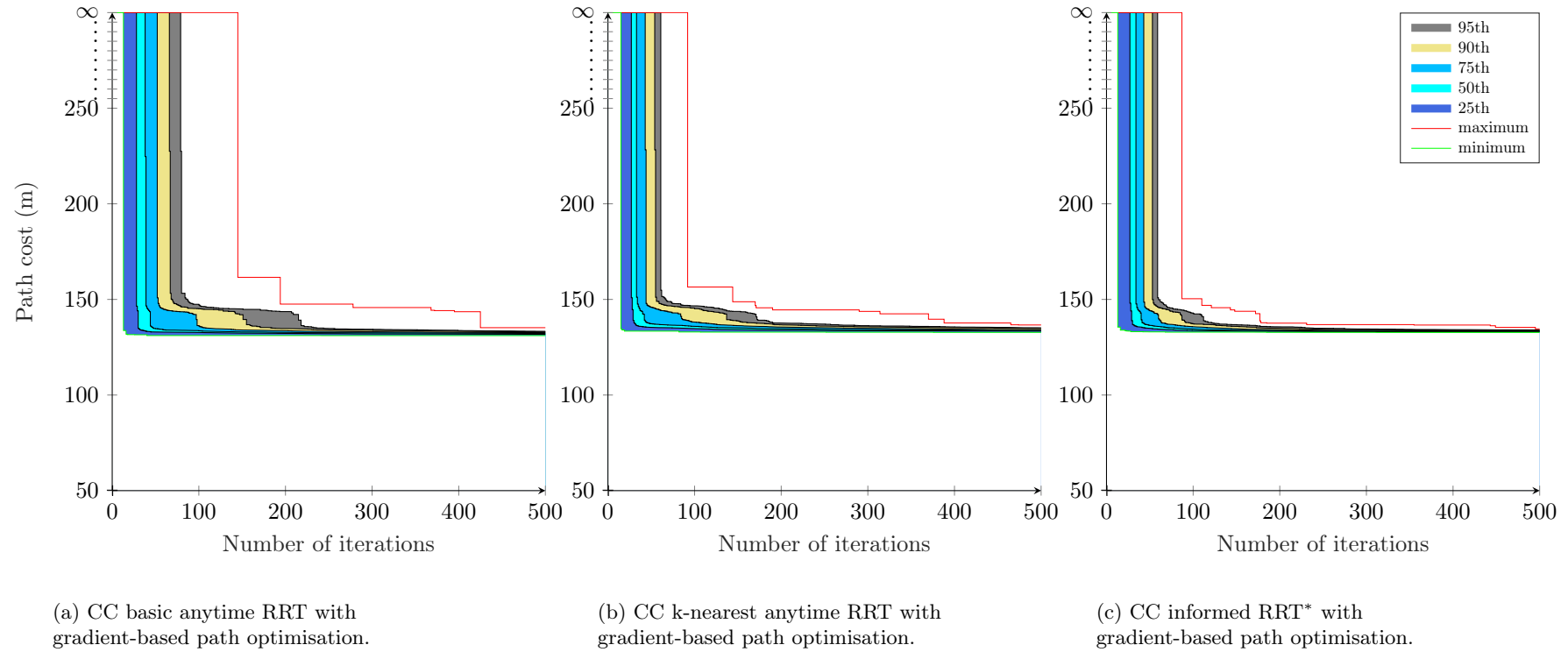


Figure 5.19: A percentile plot showing the evolution or improvement of path cost over an increasing number of iterations for 400 runs of each of the three optimised path planners formed by incorporating gradient-based path optimisation.

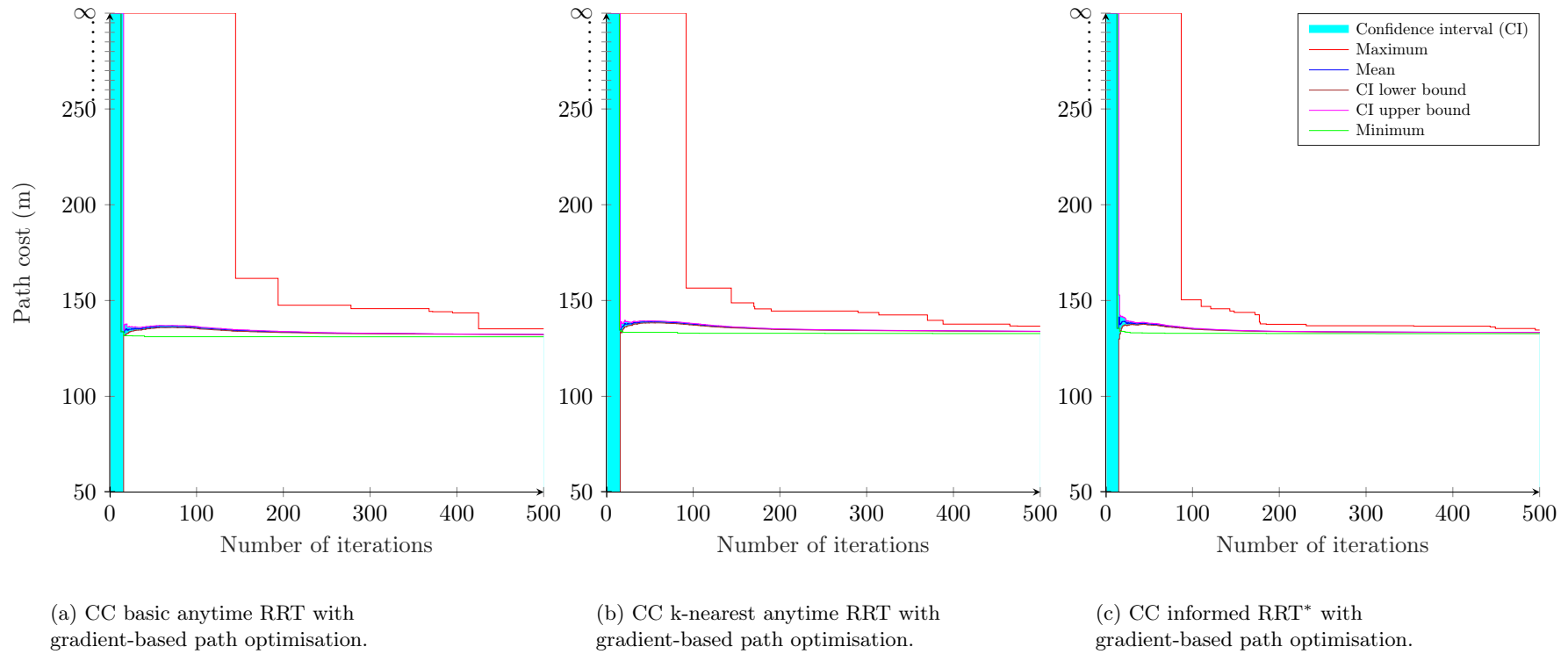


Figure 5.20: A plot of the mean path cost per iteration, with a confidence interval, for 400 independent runs of each optimised global path-planning algorithm based on gradient-based path optimisation.

	Iteration No.	Min.	Max.	25 th	50 th	75 th	IQR	90 th	95 th	Mean	Std. dev.
optimised CC basic anytime RRT	50	131.100	∞	132.963	134.688	∞	∞	∞	∞	136.100	5.375
	100	131.100	∞	132.699	133.577	136.0720	3.373	145.131	147.074	135.921	5.134
	150	131.100	161.553	132.385	132.981	133.974	1.589	141.992	145.000	134.332	4.115
	200	131.100	147.551	132.216	132.765	133.656	1.440	134.768	143.631	133.551	2.942
	250	131.065	147.551	132.096	132.584	133.243	1.147	134.025	134.794	133.017	2.169
	300	131.065	145.745	131.975	132.488	133.070	1.095	133.844	134.391	132.794	1.728
	350	131.065	145.745	131.927	132.431	132.947	1.020	133.661	134.135	132.657	1.558
	400	131.065	143.564	131.864	132.350	132.779	0.915	133.326	133.739	132.426	0.899
	450	131.065	135.219	131.802	132.264	132.670	0.868	133.152	133.501	132.308	0.667
	500	131.065	135.219	131.733	132.176	132.622	0.889	133.047	133.326	132.240	0.642
optimised CC k-nearest anytime RRT	50	133.406	∞	135.376	137.292	145.861	10.485	∞	∞	138.867	4.941
	100	132.973	156.483	134.705	135.883	138.110	3.405	145.167	146.627	137.555	4.325
	150	132.901	148.781	134.178	135.101	136.293	2.115	138.747	143.724	135.914	2.855
	200	132.901	144.517	133.945	134.710	135.538	1.593	136.673	137.642	135.017	1.695
	250	132.901	144.517	133.757	134.330	135.195	1.438	136.066	137.100	134.635	1.345
	300	132.837	143.736	133.662	134.137	134.884	1.222	135.682	136.249	134.413	1.125
	350	132.837	142.466	133.607	134.014	134.683	1.076	135.379	135.936	134.243	0.989
	400	132.759	137.681	133.555	133.922	134.457	0.902	135.188	135.577	134.092	0.783
	450	132.759	137.681	133.489	133.831	134.250	0.761	134.970	135.379	133.987	0.711
	500	132.759	136.631	133.440	133.774	134.156	0.716	134.764	135.145	133.893	0.627
optimised CC informed RRT*	50	133.011	∞	134.837	136.406	144.076	9.239	∞	∞	137.701	4.359
	100	132.881	150.356	133.693	134.424	135.686	1.993	137.205	142.592	135.233	2.665
	150	132.881	143.797	133.449	133.912	134.659	1.210	135.869	136.737	134.302	1.434
	200	132.785	137.584	133.335	133.671	134.165	0.830	134.915	135.684	133.870	0.815
	250	132.764	136.827	133.196	133.486	133.900	0.704	134.376	134.915	133.653	0.660
	300	132.764	136.827	133.152	133.421	133.779	0.627	134.188	134.546	133.543	0.570
	350	132.720	136.827	133.111	133.352	133.674	0.563	134.050	134.325	133.462	0.526
	400	132.720	136.601	133.093	133.300	133.615	0.522	133.952	134.199	133.405	0.460
	450	132.720	135.389	133.063	133.266	133.523	0.460	133.855	134.097	133.343	0.370
	500	132.720	134.598	133.054	133.238	133.478	0.424	133.769	134.003	133.309	0.340

Table 5.5: Numerical summary of the statistics of the optimised path planners based on gradient-based path optimisation, at selected number of iterations.

the other hand, the maxima of the optimised CC k-nearest anytime RRT and the optimised CC informed RRT* are respectively up to 12.72% and 13.46% smaller than their baselines.

Considering the percentiles, we observe that the optimised CC basic anytime RRT's 25th and 50th percentiles are consistently below the same percentiles of the other two optimised path planners. Also, from 250 iterations onwards, the optimised CC basic anytime RRT's 75th percentile is consistently below the corresponding percentiles of the other two optimised path planners. Lastly, we note that the CC basic anytime RRT's 90th and 95th percentiles are consistently below the maxima of the other two optimised path planners.

With regard to the mean path cost, the optimised CC basic anytime RRT's mean is consistently below the mean path costs of the two other optimised path-planning algorithms. It is followed by the optimised CC informed RRT*, whose mean path cost is consistently below that of the optimised CC k-nearest anytime RRT. From 200 iterations onwards, the optimised CC basic anytime RRT's mean path cost consistently stays below the optimised CC k-nearest anytime RRT's 25th percentile; then from 250 iterations onwards, the optimised CC basic anytime RRT's mean path cost consistently stays below the optimised CC informed RRT*'s 25th percentile. In comparing the mean path cost of the optimised CC basic anytime RRT with the minima of the other two optimised path-planning algorithms, we find that from 300 iterations onwards, it consistently stays below the minimum of the optimised CC k-nearest anytime RRT, and from 350 iterations onwards, it consistently stays below the optimised CC informed RRT*'s minimum. In comparison to its baseline, the optimised CC basic anytime RRT's mean path cost is up to 20.40% smaller. On the other hand, the mean path costs of the optimised CC k-nearest anytime RRT and the optimised CC informed RRT* are respectively only up to 7.64% and 6.68% smaller than their baselines.

Evaluating the three optimised path planners based on gradient-based path optimisation based on the above analyses, it is clear that, again, the two contending algorithms are the optimised CC basic anytime RRT and the optimised CC informed RRT*, as was the case in Subsections 5.6.3.2 and 5.6.3.3. Without doubt, the optimised CC basic anytime RRT does well in terms of the minimum, the mean path cost, as well as the 25th and 50th percentiles. This time, the optimised CC basic anytime RRT clearly comes second with regard to the maximum cost, with the optimised CC informed RRT* being the first. For the 75th, 90th and 95th percentiles, adjudication is required. In early iterations, for the optimised CC informed RRT*, these percentiles are consistently below those of the optimised CC basic anytime RRT. However, in later iterations, the optimised CC basic anytime RRT takes the lead with respect to these percentiles. Again, considering the fact that the optimised CC basic anytime is generally faster, it may not necessarily be set back by the fact that it only finds low-cost values for these percentiles in later iterations.

Thus, considering the number of considered criteria in which the optimised CC anytime RRT leads the optimised informed RRT*, we can consider the optimised CC basic anytime RRT to be significantly improved by the incorporation of the gradient-based path-optimisation step.

5.7 Optimised Global Path Planning Summary

This chapter was focused on the development of algorithms that solve the optimised path planning problem. These are path-planning algorithms that are not only capable of finding a path from the initial configuration, \mathbf{q}_I , to the goal configuration, \mathbf{q}_G , that is feasible, but one that also satisfies some given optimality criteria. In this thesis, we are interested in the minimisation of path length.

For this purpose, three benchmark path planners, namely the CC basic anytime RRT, the CC k-nearest anytime RRT, and the CC informed RRT*, have been developed. These path planners are capable of planning paths that are curvature continuous. All three benchmark path planners strive towards the optimal solution by continually shrinking the search space by bounding it using the cost of each newly-found solution – a process that encourages each search to find a better solution than all previously-found solutions. The difference between the benchmark path planners lies in the quality of the solutions they find and the amount of time they take to compute these solutions. At one end is the CC basic anytime RRT, which produces solutions quickly, though the returned solutions are almost-surely suboptimal. At the other end is the CC informed RRT*, which is capable of finding the optimal solution as the number of iterations tends to infinity, i.e. asymptotically optimal. The CC k-nearest is in-between these two extremes. Four path-

optimisation algorithms were then developed for the purpose of accelerating the convergence of the benchmark path planners. Each of these path-optimisation algorithms has been incorporated into the benchmark path planners to reduce the cost of each newly-found solution before it is used to bound the search space. This resulted in four sets of optimised path planners, with each set corresponding to optimised planners formed by incorporating one of the path-optimisation algorithms into each of the benchmark path planners. Each set of optimised path planners has been analysed and compared to the benchmark path planners in the bid to study the effectiveness of each path-optimisation algorithm in accelerating the convergence of the benchmark path planners. An important aspect of this study was to ascertain if incorporating path-optimisation algorithms can help a quick, almost-surely suboptimal path planner, like the CC basic anytime RRT, to attain comparable or better performance than asymptotic, almost-surely optimal path planners like the CC informed RRT*. From the analyses presented in Subsections 5.6.3.1, 5.6.3.2, 5.6.3.3 and 5.6.3.4, it is clear that incorporating a path-optimisation step to the benchmark path planners accelerates their convergence. Incorporating a path-pruning step was found to mostly benefit the CC basic anytime RRT, with only minor improvements to the CC k-nearest anytime RRT and CC informed RRT*. This results in an optimised CC basic anytime RRT, whose performance is comparable to the CC k-nearest anytime RRT and the asymptotically optimal CC informed RRT*. However, with path pruning, clear convergence is not achieved. In contrast, incorporating random shortcut, the wrapping process and gradient-based path optimisation to the benchmark path planners results in clear convergence towards the minimum for all the optimised path-planning algorithms. Since the CC basic anytime RRT is quick to find solutions, it allows for the incorporation of path optimisation without incurring too much computation time. Moreover from the analyses, the optimised CC basic anytime RRT led the optimised CC k-nearest anytime RRT and optimised CC informed RRT* in most considered criteria. This suggests that using a path-optimisation step to accelerate the convergence of the benchmark path planners does help the almost-surely suboptimal path planner, the CC basic anytime RRT, to attain comparable or better performance than the asymptotic, almost-surely optimal path planner, the CC informed RRT*.

With the ability to plan optimised paths in place, the next step is to equip an autonomous vehicle with the ability to accurately execute such paths. The development of a path-tracking controller that serves this purpose is the subject of the next chapter.

Chapter 6

Path-tracking Controller Design and Implementation

Once continuous-curvature paths have been planned and optimised, the next step, which completes the desired outcome of autonomous navigation, is path tracking. Path tracking refers to the robot's ability to follow the planned paths. This chapter details the design and implementation of a path-tracking controller, whose objective is to follow continuous-curvature paths generated by the path-planning algorithms developed in the previous chapter.

The path-tracking controller takes as input the planned path as well as the robot's state information and it outputs motion commands that the robot has to execute to stay on the planned path or return to it in case of deviation. Since the path is given beforehand, it is possible to use information from this path to compute feed-forward control signals, which can cause the robot to perfectly follow the planned path in the absence of disturbances. However, since disturbances are inevitable in practical systems due to issues ranging from imperfect state measurements, imperfect actuators, to environment-related issues such as uneven surfaces, feed-forward control is practically insufficient and there is a need for feedback control. In Chapter 3, we conducted a review of existing path-tracking techniques and selected those that are suitable for adaptation and extension in the design of the feed-forward and feedback components of the path-tracking controller designed in this chapter.

Central to the task of path tracking is the need to know the position and orientation (pose) of the robot with respect to the planned path and any other objects in the environment. It is thus important to model this relationship between the robot and its environment before designing the controller. The remaining parts of this chapter present this modelling of the robot and the environment in which it operates (Section 6.1) as well as the design of the path-tracking controller (Section 6.2).

6.1 Plant Modelling

Before we proceed to the design of control laws to achieve path tracking, we first introduce how the robot and the world in which it operates are modelled. Two axis systems, namely the inertial and body axes are defined in Subsection 6.1.1. In essence, they are respectively used to relate the states of the robot in relation to the environment and itself. In addition to this, the dynamics describing how the available control inputs affect the motion of the robot are then discussed in Subsection 6.1.2. In the same subsection, sources of the robot's state estimates are stated. These models are then used in the design of the path-tracking controller in Section 6.2.

6.1.1 Axis Systems

The first aspect of modelling the robot and the environment in which it operates involves the definition of axis systems. One of these axes is fixed to the world and another is fixed to the robot's body. While these axis systems have been earlier introduced as the inertial and body axes, they are also sometimes respectively referred to as the global and local coordinate systems. They are discussed next.

6.1.1.1 Inertial Axes

The vehicle operates in a world modelled with an axis system known as the inertial axis system. The most common conventions for this axis system in autonomous vehicles research are the north-east-down (NED) and the east-north-up (ENU), with the NED convention often used for aviation systems, while for ground robots as well as aerial robots with affixed manipulators, the ENU convention is commonly used [165]. In this project, our test vehicle is a ground robot and moreover, the simulation environment in which our algorithms are deployed uses the ENU convention. As such, we choose the ENU convention. In this convention, the axes are defined as follows: the x -axis, X_E , points in the direction of the geographic east of the earth, while the y -axis, Y_E , points towards the geographic north; the z -axis, Z_E , completes the right-handed orthogonal axis system and points upwards while being perpendicular to both the X_E and Y_E . The system designer is at liberty to choose any stationary reference point (or one with a constant velocity) in the world as the origin of the coordinate system. Unlike in the NED convention, all angles in the ENU convention are measured relative to the east axis instead of the north. An illustration of this coordinate system is shown on the left of Figure 6.1. In this example, the origin of the coordinate system is chosen to be the bottom left corner of the environment. This coordinate system forms the basis of the map of the world used for path planning. Thus the planned paths that the robot is expected to execute are defined in this coordinate system. It is also relative to this axis system that we humans perceive, describe and report the motion of the robot in the world.

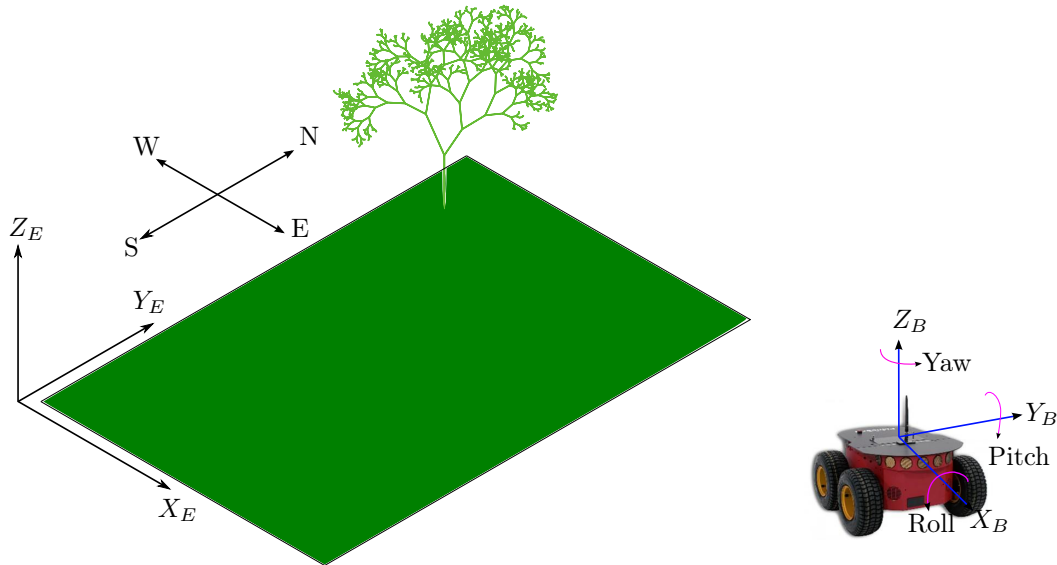


Figure 6.1: An illustration of the inertial ENU (east-north-up) and body axis system. The inertial axis system is shown on the left and the body axis system is as shown on the right. They are both right-handed orthogonal axis systems. It is important to note that unlike in the NED convention, all angles in ENU inertial system are measured relative to the east axis rather than the north.

6.1.1.2 Body Axis

Just as we have an axis system fixed on the earth's surface to describe states of objects that are contained in the world, we have an axis system fixed on the body of the robot for describing and relating entities which are on the robot. This is called the body axis system and in this thesis, it is denoted as the $X_B Y_B Z_B$ axis system. Similarly to the inertial axis system, it is also a right-handed orthogonal axis system. In this case, the origin is constrained to coincide with the centre of mass of the robot or simply placed on top of the robot. The x -axis, X_B , then points in the forward direction of the robot, while the y -axis, Y_B , points outwardly to the left of the robot while maintaining a right angle with the x -axis, and lastly, the z -axis, Z_B , points vertically upwards

while being orthogonal to both the x - and y -axis. This coordinate system is important because sensors mounted on the robot measure quantities of interest relative to the robot's body. Examples of such quantities include the distance to objects in the world as well as inertial measurements.

6.1.1.3 Coordinate Representation and Transformations

The definition of the inertial and body axis allows us to describe the position, orientation and motion of the robot using the relationship between its body axis and the inertial axis. In general, a rigid body in 3D space has six degrees of freedom (6DOF): its position, which is given by the (x, y, z) coordinates of the origin of its body axis in the inertial axis, and its orientation, which is described by three angles of rotation of its body frame with respect to the inertial frame. A commonly-used way for describing the orientation is the use of Euler angles. In the aircraft case, rotation about the axis from nose to tail is known as the *roll* while the rotation about the axis from wing to wing (resulting in nose up or down) is known as *pitch* and finally, the rotation about the axis from top to bottom (resulting in nose left or right) is known as the *yaw*. This description can be generalised to any type of vehicle, but the aircraft is convenient for illustration purposes. There are a number of conventions for representing Euler angles, depending on the order in which these rotations are carried out. The most commonly used is the Euler 3-2-1 (or Euler Z-X-Y convention) – this is the convention used in this thesis. In this convention, to get the final orientation, the body is first rotated by an angle θ (the yaw) about the body z -axis followed by a rotation by an angle ψ (the pitch) about the body y -axis and finally, a rotation by an angle ϕ (the roll) about the body x -axis.

It is important to note that for purposes of this project, the motion of the robot used (which is a wheeled ground robot) is confined to the 2D space – the ground plane. As such, the position of the robot can be represented using coordinates (x, y) , with the z coordinate always being zero. Moreover, the robot can only rotate about the body z -axis (yaw). This results in three degrees of freedom (3DOF) compared to the six degrees of freedom in the 3D case. As a result, the position and orientation of the robot in the world can be represented by the state vector $[x \ y \ \theta]$. Also, for dynamic modelling of such vehicles, only forces, moments and velocities in x and y directions as well as angular rates about the body z -axis are considered. The dynamics are discussed in the next section. We use the subscript B for vectors of forces, moments and velocities to denote that they are defined with respect to the body frame. For position coordinates, we use a superscript to specify the axis system in which they are defined, for example the position coordinate (x^E, y^E) denotes coordinate (x, y) defined in the inertial frame. In a later subsection (Subsection 6.2.2), we introduce a new axis system known as the guidance axis – the same notation is used to describe coordinates defined in this frame, with the superscript *guidance* used. In the same subsection, we also extend the notation so as to differentiate between coordinates of different entities in the same coordinate frame by making use of a subscript to specify the entity that the coordinate represents. As an example, $(x_{\text{rob}}^E, y_{\text{rob}}^E)$ denotes the (x, y) coordinate of the robot in the inertial frame.

6.1.2 Robot Dynamics and Robot State Estimates

With the axis systems discussed, there are two more things that are important to understand to be able to successfully design the path-tracking controller. The first is to understand the dynamics of the particular vehicle being controlled. It is important to understand the dynamics because it is only after we understand what paths the vehicle will follow given specific inputs that we can truly be able to design a controller that gives the right commands to cause it to follow a given path. Secondly, it is important to understand the sources from which estimates of vehicle states will be obtained as well as the format of these states. In line with this second point, it is also important to know the interfaces through which motion commands have to be passed, as well as the required format of these commands. The following two paragraphs discuss these issues.

6.1.2.1 Robot Dynamics

Dynamics is the branch of classical mechanics, which studies bodies in motion. There are two branches of dynamics, namely kinematics and kinetics. Kinematics can be thought of as the geometry of motion. It describes the motion of a body without considering the forces causing the

motion – it only relates motion variables such as linear velocity, angular velocity and position to each other over time. Kinetics, on the other hand, considers the forces acting on the object and it relates these forces to the kinematic state of the object (i.e. its position, velocity and acceleration) using Newton’s laws of motion.

Figure 6.2 shows a block diagram of the 3DOF dynamics of a moving body. This block diagram shows the inputs and outputs to kinetics and kinematics blocks and the interaction between them. Inputs to the kinetics block of the system are the forces (\mathbf{F}_B), moments (\mathbf{M}_B), mass of the object (m) and the moment of inertia (\mathbf{I}_B), and the outputs are the linear and angular velocities (\mathbf{v}_B and $\boldsymbol{\omega}_B$ respectively). These output velocities are also fed back to the kinetics block. The B subscripts signify that these quantities are defined in the body axis system. The output linear and angular velocities are then fed to the kinematic block, which then outputs the position (\mathbf{P}) and the orientation (Euler angles, \mathbf{e} – in this case only the yaw) of the vehicle. The orientation is also fed back to the kinematic block. The kinematics of the vehicle are computed in inertial coordinates; as such, the linear and angular velocities are transformed from the body frame to the inertial frame before being used in the computation. The angular rate coordinates are converted to the inertial frame by using Euler 3-2-1 dynamics. On the other hand, the conversion of linear velocities from the body frame to the inertial frame is achieved through the so-called direction cosine matrix (DCM) – a standard transformation matrix that transforms one reference frame to another.

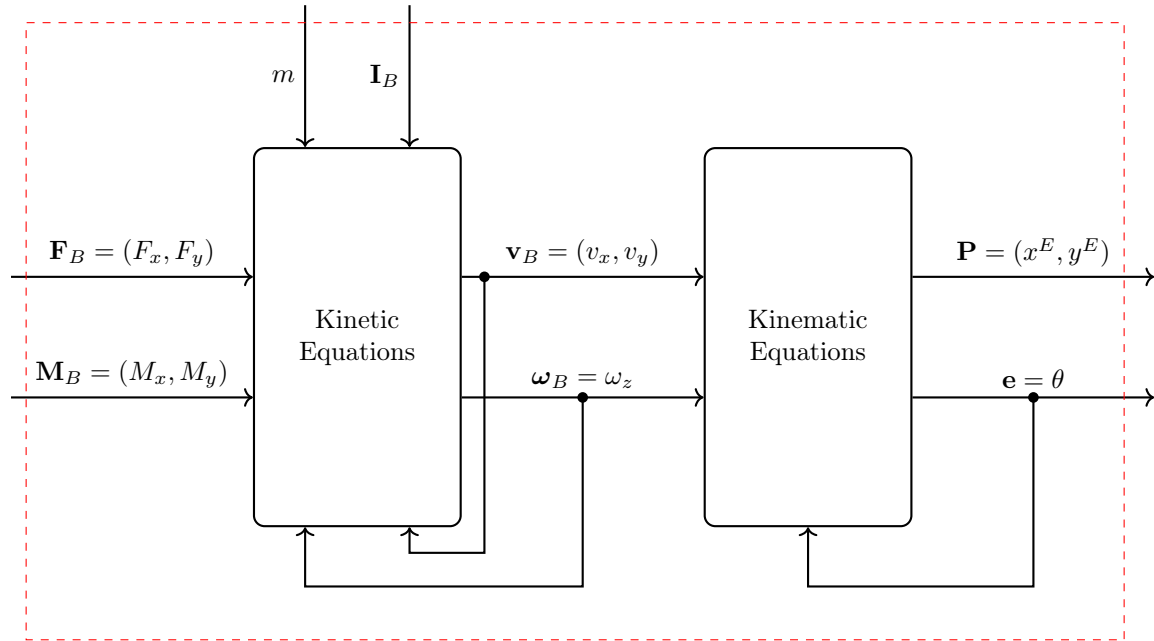


Figure 6.2: Block diagram for 3DOF dynamics of motion

For purposes of path tracking, control of a mobile robot can be achieved using either the full dynamic model or using only kinematics. If we have a way of commanding the linear and angular velocities directly, it is possible to use only the kinematic model. The kinematic model is chosen for the following reasons:

- It is simpler than the full dynamic model in that it does not rely on the knowledge of numerous parameters associated with the vehicle and its actuators, including masses, moments of inertia and torque coefficients.

- The mobile robot controlled in this project comes equipped with low-level velocity control loops that take the desired linear and angular velocities, and control the robot's motors to produce torques that cause the robot to move at the desired velocities.
- As pointed out in the literature review, path trackers based on the kinematic model are simpler to understand and implement, yet they have been proven to be effective through numerous successful autonomous vehicle project.

6.1.2.2 Kinematic Model

In Subsection 3.1 of the literature review, the kinematic model has been encountered in the form of the kinematic bicycle model. This model simplified a vehicle with steerable front wheels by collapsing the wheels in each of the axles into two wheels, located at the centre of each axle – like a bicycle. In this subsection, we consider the kinematic model of the specific robot used in this the thesis – a four-wheel skid-steered mobile robot. The robot is modelled as a point mass that is located at the its centre of mass, $(x_{\text{rob}}^E, y_{\text{rob}}^E)$. This position coincides with the origin of the body axis. The robot has a yaw angle θ with respect to the inertial x -axis. The variables v_x and ω_z denote the linear and angular velocity of the robot along the body x -axis and about the body z -axis respectively. The kinematic model of the robot is shown in Figure 6.3 and the accompanying equations are given by Equation 6.1. This is the unicycle model of a wheeled mobile robot. It is common practice in mobile robotics to design controllers based on this unicycle model and make use of a mapping from the unicycle inputs to the actual robot model inputs (e.g. the left and right wheel velocities of a differential-drive or skid-steered robot). The reason is that it is more natural to think of the path that the robot takes given inputs of the unicycle model (translational velocity and angular velocity) than it is with individual wheel velocities given. The robot used in this project provides an interface for unicycle inputs and implements the mapping from the unicycle inputs to the individual wheel velocities.

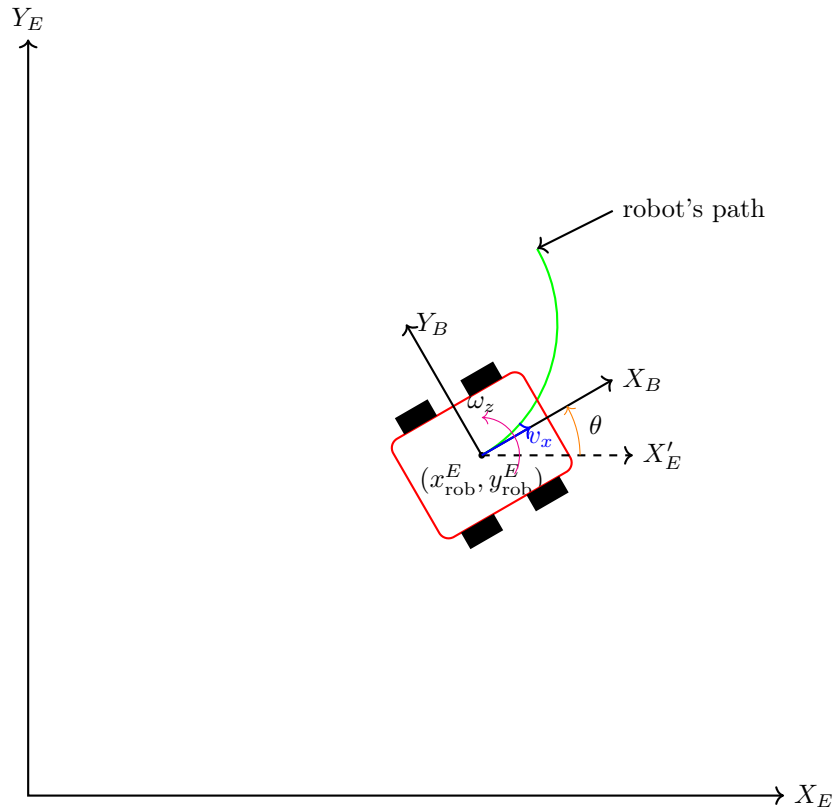


Figure 6.3: The kinematic model of the robot.

The kinematic equations describing how the robot's state changes over time are as follows:

$$\begin{aligned}\dot{x}_{\text{rob}}^E &= v_x \cos \theta, \\ \dot{y}_{\text{rob}}^E &= v_x \sin \theta, \\ \dot{\theta} &= \omega_z,\end{aligned}\tag{6.1}$$

where x_{rob}^E , y_{rob}^E and θ are the robot's states, and v_x and ω_z are control inputs. The model essentially captures the assumption that under nonholonomic constraints, the mobile robot moves in the direction of the body x -axis. For a given combination of v_x and ω_z over a period of time Δt , the robot follows a circular arc of radius, $r = \frac{v_x}{\omega_z}$.

6.1.2.3 State Estimates

To achieve path tracking, in addition to knowing the motion model of the robot, we need a way of knowing the current robot states, including position and orientation. These current states $\mathbf{q}_k = [x_k^E \ y_k^E \ \theta_k]^T$ are used to compute the error signal for the controller by comparing them with the desired state – in this case a desired configuration, $\mathbf{q}_d = [x_d^E \ y_d^E \ \theta_d]^T$ on the planned path. In this project, estimates of current states of the robot (and the simulated robot) are obtained from an odometry module. An example screenshot of the odometry information obtained from this module is shown in Figure 6.4. The position is presented as a 1×3 column vector of its 3D coordinates, while orientation is represented as a quaternion. Therefore, for consistency with our coordinate representation and transformation, wherein orientation was represented in terms of Euler angles, it will be necessary to convert the orientation from quaternion to Euler representation in the implementation of the path-tracking controller.

```
pose:
pose:
position:
x: -0.280729315238
y: 0.038176400062
z: 0.0
orientation:
x: -0.000777686476133
y: -0.00141868517546
z: -0.317729730445
w: 0.948179941218
```

Figure 6.4: An example screenshot of the robot's current state.

With the model of the environment and that of the robot as well as the source and format of state estimates discussed, we proceed to apply this information in the design and implementation of the path-tracking controller.

6.2 Path-tracking Controller Design

The task of the path-tracking controller is to cause the robot to follow a known input path received from the optimised path planner. It has to do so by computing linear and angular velocity commands that keep the robot on the path or cause it to return to the path in case of deviation. As stated in the introductory part of this chapter, it is possible to extract curvature information from input path and use it to compute feed-forward velocity commands that can cause the vehicle to perfectly follow the path in the absence of disturbances. Since disturbances are inevitable in practical systems, it is noted that feed-forward control alone cannot be enough – hence the need for feedback control. In this scheme, feed-forward control attempts to make the robot to follow the nominal path, minimising the amount of tracking errors to be corrected by feedback control. This control architecture is shown in Figure 6.5. In Chapter 3 of the literature review, we explored available path-tracking controllers and the outcome of that review was the identification of existing

path trackers that are suitable for adaptation and extension in the design of the feed-forward and feedback components of the path tracker designed in this thesis. We discuss the adaptations of the ideas from literature to the design of our feed-forward and feedback controllers in Subsections 6.2.1 and 6.2.2.

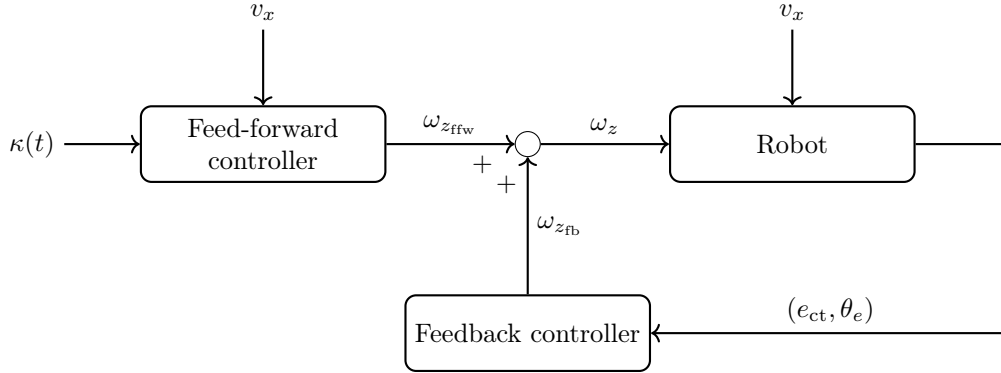


Figure 6.5: Controller architecture. The feed-forward controller uses the curvature profile of the planned path, represented by $\kappa(t)$, together with the robot's current linear velocity to produce the feed-forward angular velocity, $\omega_{z_{ffw}}$. The feedback controller uses the cross-track error, e_{ct} , and the heading error, θ_e , together with the robot's current pose to compute the feedback angular velocity, $\omega_{z_{fb}}$.

6.2.1 Feed-forward Control

The task of the feed-forward controller is to control the robot so that it follows the planned path in an open-loop manner. Among the existing path trackers encountered in the literature review, the continuous-curvature (CC) path tracker was selected as a suitable feed-forward path tracker for our project. In the application found in literature, the path tracker was used to follow paths composed of a concatenation of straight lines. It works by computing a CC path that starts from the robot's position and ends at a point that lookahead distance ahead of the robot on the path; the robot is then made to follow the generated CC path by using its curvature profile to deduce appropriate motion commands. In the original context, the path tracker worked as a pursuit-based path tracker because it could not explicitly follow the planned discontinuous-curvature path, but aimed to pursue the ever-moving lookahead point. However, in this project, it is used to explicitly track the planned path in an open-loop fashion. More precisely, since the paths planned in this thesis are curvature continuous and have a curvature profile that is within the robot's kinematic capabilities, then applying the commands derived from the planned path's curvature profile will cause the robot to accurately follow the planned path in the absence of disturbances; thus serving as a feed-forward controller.

It is important not to overlook practical issues with controller implementation. One of these issues is discretisation, i.e. the fact that we can only be able to apply control commands at sampling intervals and hold them constant over the sampling interval. The existence of such issues means that even in the absence of disturbances, there will still be slight inaccuracies with feed-forward control. Due to this reason, the need for feedback control is inevitable. We now proceed to the design of the feedback controller.

6.2.2 Cross-track Error Control

In the presence of disturbances and imperfect actuators, the feed-forward path-tracking control introduced in the previous section will not be enough for accurate path-tracking; moreover, even in the absence of disturbances, slight inaccuracies will still be incurred due to inherent issues in control design, such as discretisation. In either case, it is the purpose of the cross-track error

controller designed in this subsection to correct for the deviation by controlling the cross-track position error to zero, causing the robot to stay on the planned path.

Just as the literature review succeeded in identifying a suitable technique for the feed-forward path-tracking controller for adaptation, suitable techniques were also identified for feedback path-tracking control. These are position-based feedback path trackers. These techniques work by measuring the cross-track error, which is the shortest distance from a reference on the robot to the path being tracked, and then using the cross-track error together with the robot's velocities to design a feedback control law that causes the robot to track the planned path. Among considered path trackers, the front-wheel position-based path tracker was found to be most effective, followed by the rear-wheel position-based feedback path tracker. In the design of our cross-track error path tracker, we draw inspiration from these path trackers, and apply it to design a cross-track error controller that suits our robot.

The design of the cross-track error controller begins with being able to relate the robot to the planned path. Without this ability, it would be impossible to compute the cross-track error, let alone the design of the feedback control law. The path being tracked in this project is a continuous-curvature path generated by the optimised path-planning algorithms developed in Chapter 5. Such a path is made up of a concatenation of continuous-curvature path primitives including clothoids, circular arcs and straight lines. It is output by the global path planner as a series of configurations in the configuration space, \mathcal{C} , with each configuration represented as a coordinate (position and orientation) in the inertial axis system, at that coordinate. It is important to note a difference in terminology between path-planning and path-tracking literature. Path-planning literature refers to points along the planned path as configurations, while path-tracking literature refers to them as waypoints. Henceforth, we refer to a configuration, \mathbf{q} , on a planned path as a waypoint. A path with n waypoints is represented as $\{\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3, \dots, \mathbf{q}_n\}$, where each waypoint $\mathbf{q}_i = [x_i^E \ y_i^E \ \theta_i]^T$. As explained in Subsection 4.2.1, the path is composed of primitive manoeuvres. Each waypoint, \mathbf{q}_i ($i > 1$), has a path primitive (clothoid, circular arc, or straight line), by which it is reached from the preceding waypoint, \mathbf{q}_{i-1} , associated with it. The task of the cross-track error controller is to accurately follow each of these path primitives, starting from the first waypoint until the last. We explain the process by which this goal is achieved, starting with the case of tracking straight-line path segments. The same process is used for tracking the other path primitives, with the difference being in how the cross-track error is computed.

The model of the cross-track error controller implemented in this project is as shown in Figure 6.6. The controller works by picking two successive waypoints on the path, $\mathbf{q}_{\text{source}} = (x_{\text{source}}^E, y_{\text{source}}^E)$ and $\mathbf{q}_{\text{dest}} = (x_{\text{dest}}^E, y_{\text{dest}}^E)$, starting from the beginning of the path. For each such waypoint pair, the subpath connecting them, also known as the *track*, is the path primitive associated with \mathbf{q}_{dest} . Starting with the case of a straight-line track (illustrated in Figure 6.6), important attributes of the track are its length, l_{track} , and its orientation with respect to the inertial x -axis, θ_{track} . From the source and destination waypoints, expressions for these attributes of the straight-line track can be respectively computed as:

$$\theta_{\text{track}} = \arctan \left(\frac{y_{\text{dest}}^E - y_{\text{source}}^E}{x_{\text{dest}}^E - x_{\text{source}}^E} \right), \quad (6.2)$$

$$l_{\text{track}} = \sqrt{(x_{\text{dest}}^E - x_{\text{source}}^E)^2 + (y_{\text{dest}}^E - y_{\text{source}}^E)^2}.$$

The desired robot position on the track is the projection of the robot onto the track. In Figure 6.6, this is marked \mathbf{q}_d – the desired waypoint. This is a virtual waypoint that progresses along the track as the robot moves towards \mathbf{q}_{dest} . With \mathbf{q}_d known both the in-track distance and the cross-track error can be calculated. The in-track distance is simply the distance from $\mathbf{q}_{\text{source}}$ to \mathbf{q}_d , while the cross-track error, e_{ct} , is given by the perpendicular distance from the robot's location to the track. A convenient trick that simplifies the calculation of these parameters relies on the definition of an auxiliary coordinate system known as the *guidance axis*, represented by the $X_{\text{guidance}}-Y_{\text{guidance}}$ axes in Figure 6.6. To obtain the guidance axes, we first rotate the inertial axes by the track orientation θ_{track} , and then translate its origin so that it coincides with the source waypoint, $(x_{\text{source}}^E, y_{\text{source}}^E)$. Using the guidance axis, to obtain the in-track distance and the cross-track error, the position of the robot in inertial coordinates, $(x_{\text{rob}}^E, y_{\text{rob}}^E)$, is first transformed to guidance coordinates:

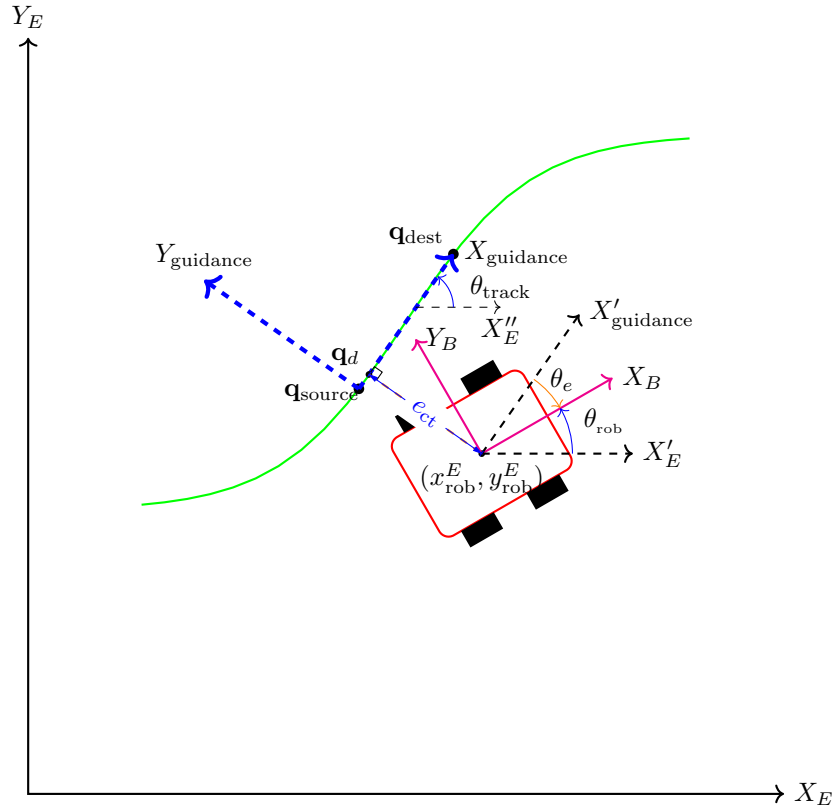


Figure 6.6: The model used in this project for the cross-track error control process.

$$\begin{bmatrix} x_{\text{rob}}^{\text{guidance}} \\ y_{\text{rob}}^{\text{guidance}} \end{bmatrix} = \begin{bmatrix} \cos \theta_{\text{track}} & \sin \theta_{\text{track}} \\ -\sin \theta_{\text{track}} & \cos \theta_{\text{track}} \end{bmatrix} \begin{bmatrix} x_{\text{rob}}^E - x_{\text{source}}^E \\ y_{\text{rob}}^E - y_{\text{source}}^E \end{bmatrix}. \quad (6.3)$$

The beauty of this transformation is that once it is done, the cross-track error is given by the y -component of the robot's position in the guidance axis system, $y_{\text{rob}}^{\text{guidance}}$, and the in-track distance is simply the x -component, $x_{\text{rob}}^{\text{guidance}}$.

The preceding discussion has explained the process of computing the cross-track error and in-track distance for straight-line tracks. It is important to note that for the straight-line track, the orientation of all points on the track is the same. This is not the case of a clothoid and a circular arc. Also, the length of the track is no longer as given in Equation 6.2, but rather, the length of the corresponding curve. This means that the process used to compute the cross-track error and the in-track distance for straight-line tracks is not applicable to these cases. We illustrate the computation of the cross-track error and in-track distance for the case of a circular track. This calculation is also used for a clothoid, with the clothoid approximated by an arc spline using the techniques of Meek et al. [166], since it has no closed form. The calculation of the cross-track error and in-track relies on the two end points of the circular track ($\mathbf{q}_{\text{source}}$ and \mathbf{q}_{dest}), as well as its centre and its radius (both computed at planning). From Figure 6.7, the cross-track error is given by difference between the robot's radial distance from the centre of the track and the radius of the track:

$$e_{\text{ct}} = \sqrt{(x_{\text{rob}}^E - x_{\text{cTrack}}^E)^2 + (y_{\text{rob}}^E - y_{\text{cTrack}}^E)^2} - r_{\text{track}}, \quad (6.4)$$

where $(x_{\text{cTrack}}^E, y_{\text{cTrack}}^E)$ is the centre of the track in inertial coordinates and r_{track} is the track radius. The in-track distance is given by the length of arc from $\mathbf{q}_{\text{source}}$ to the desired robot position on the track, \mathbf{q}_d ; this time \mathbf{q}_d is simply the point that is a distance r_{track} from the track's centre in

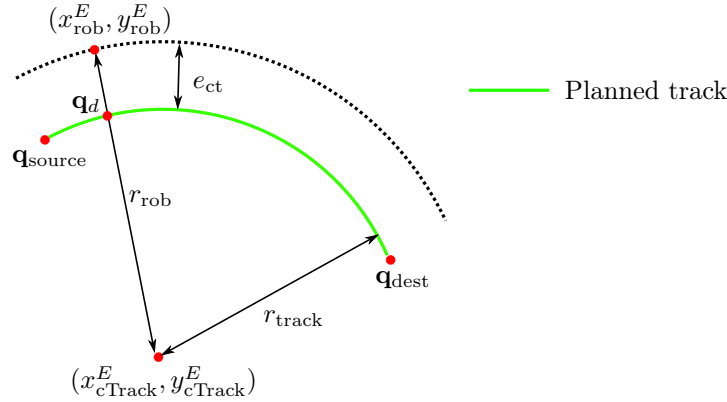


Figure 6.7: Calculation of the cross-track error and in-track distance for a circular track.

the direction of the robot. Lastly, the track length, l_{track} , is simply the arc length of circular arc – it is computed at planning.

With the cross-track error and in-track distance computations defined, the remaining task is to use these parameters in a control scheme that compensates for the cross-track error. When following a track between two waypoints, the robot is on-track if its orientation, θ_{rob} , equals the orientation of the track at that point, θ_{track} (i.e. the orientation error, θ_e , equals zero), and the cross-track, e_{ct} , error is zero. Once the in-track distance equals the track length, l_{track} , the robot has reached the destination waypoint, \mathbf{q}_{dest} – at this point, the controller needs to select a new waypoint pair for the new track. This selection of the next waypoint is the task of a *waypoint scheduler*. This process continues until the robot reaches the end of the planned path.

Figure 6.8 shows the cross-track error model of Figure 6.6 updated to illustrate the dynamics of the cross-track error. These dynamics are represented by the cross-track error rate, \dot{e}_{ct} . The figure shows that if the robot moves at constant linear velocity, v_x , and with a constant heading relative to the track (i.e v_x and θ_e constant), then the cross-track error rate is:

$$\dot{e}_{\text{ct}} = v_x \sin \theta_e. \quad (6.5)$$

With small angle assumptions, Equation 6.5 can be linearised, by realising that $\sin \theta_e \approx \theta_e$, to get:

$$\dot{e}_{\text{ct}} = v_x \theta_e. \quad (6.6)$$

With the dynamics of the cross-track error derived and linearised, we are ready to formulate a control law that aims at keeping the cross-track error at zero. We wish to correct for the cross-track error by commanding a correction angular velocity, ω_{z_c} , so that this correction turning rate modifies the current turning rate of the robot in a manner that reduces the cross-track error. This is illustrated in the block diagram of Figure 6.9. Here, the plant is the kinematic model of the robot with inputs v_x and ω_z . The linear velocity, v_x , is held constant and $\omega_{z_{\text{current}}}$ represents the current turning rate of the robot. The input to the controller is the error between the desired cross-track error (zero) and the measured cross-track error. Using this error signal, the controller computes the correction angular velocity, ω_{z_c} , which is then used to adjust the current turning rate, resulting in an angular velocity command ω_z that is then sent to the robot.

From the block diagram, the transfer function of the plant is:

$$G(s) = \frac{\dot{e}_{\text{ct}}(s)}{\omega_{z_c}(s)}. \quad (6.7)$$

From the kinematic equations of Equation 6.1, it is straightforward that $\omega_{z_c}(t) = \dot{\theta}_e(t)$. Substituting this expression and the expression of the cross-track error rate from Equation 6.5 into Equation 6.7, the transfer function of the open-loop system is:

$$G(s) = \frac{v_x}{s^2}. \quad (6.8)$$

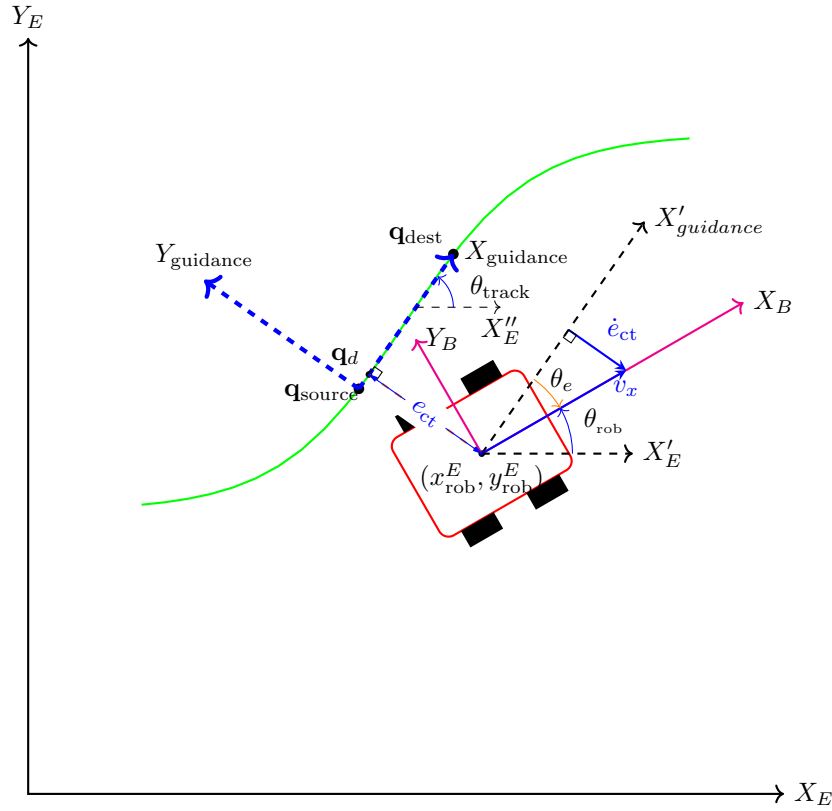


Figure 6.8: An illustration of the cross-track error rate

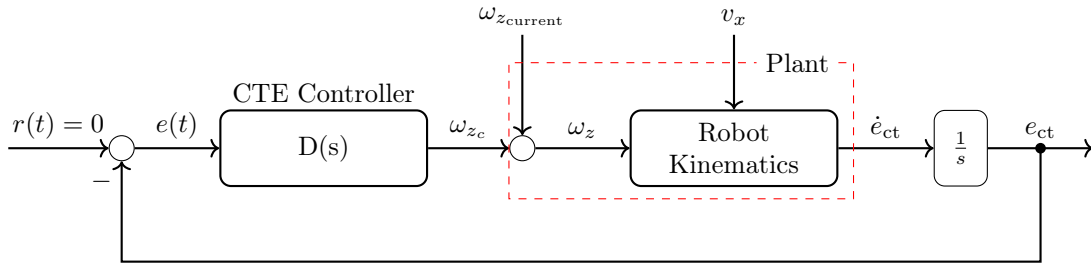


Figure 6.9: Cross-track error controller conceptual block diagram

This open-loop system has two poles at the origin and is unstable, with a step response shown in Figure 6.10. The design of a controller to stabilise the system follows next.

The control architecture shown in Figure 6.9 makes use of the cross-track error to close the loop of the system. This was for purposes of illustration of the general idea of the controller. With single-loop proportional control, the system becomes marginally stable for all gain values, with the root locus and step response shown in Figure 6.11. It is therefore necessary to employ a control strategy that shapes the root locus of the system, making it possible for the closed loop poles to move into the left-half plane. For this purpose, we make use of cascade control – where the control system consists of two feedback loops: an inner and an outer feed-back loop. The outer-loop (or primary) controller acts on the primary controlled variable (in this case, the cross-track error), while the inner-loop (or secondary) controller controls a secondary controller variable which acts as an early warning for possible changes in the primary controlled variable. In our case, the secondary variable is the cross-track error rate. A block diagram of the resulting cross-track error controller is shown in Figure 6.12. Generally, the inner- and outer-loop controllers, D' and D'' , can take any

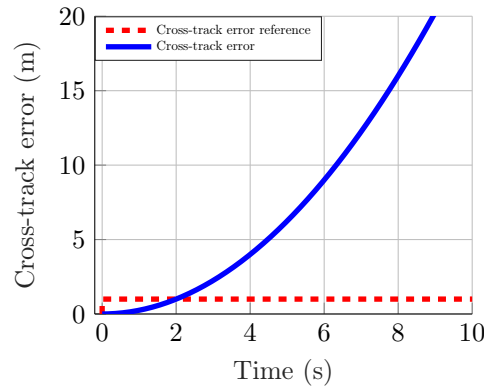


Figure 6.10: A step response of the unstable open-loop system.

form, but we found proportional cascade to be effective in this application. The next step is thus to design the proportional controller gains, k' and k'' ; we used root locus design for this purpose. We outline the process followed next.

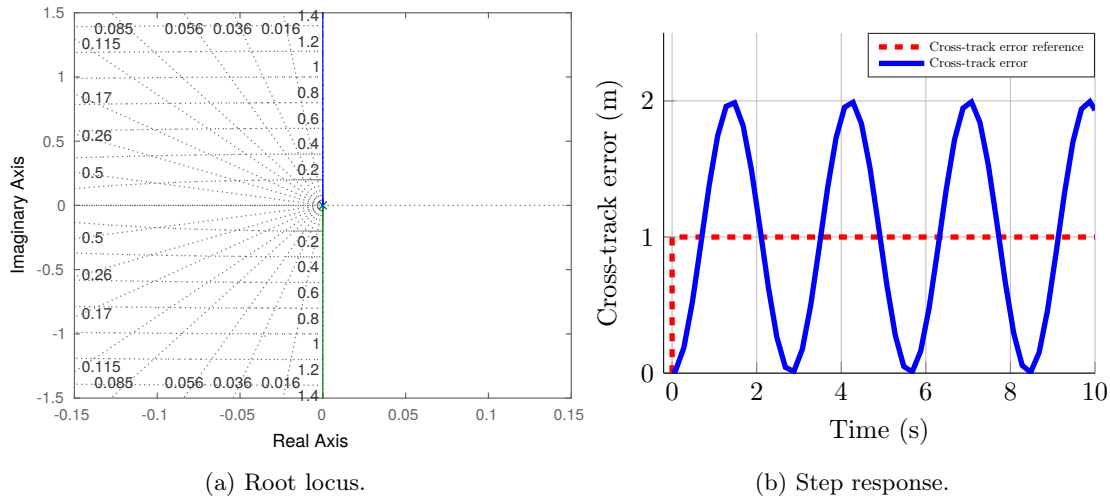


Figure 6.11: Root locus and step response of the system with single-loop proportional control. The system becomes marginally stable no matter the value of the gain.

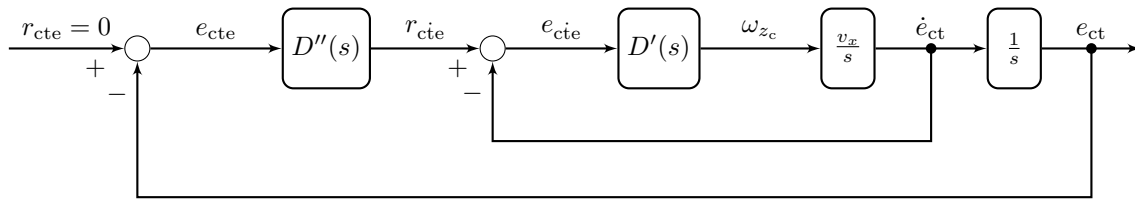


Figure 6.12: Block diagram of the cross-track error controller

Recommended practice in cascade controller design is to first design the inner loop-controller and then design the outer-loop controller. It is recommended to design these controllers such that the inner-loop controller is much faster than the outer-loop controller, e.g. at least 3 times

faster [167]. From a high-level viewpoint, this translates to a settling time of at least a third of that of the outer-loop controller. In our design, we set controller specifications based on the following requirement: we desire that if the robot has been offset from the planned path by a distance of up to 1 m, the controller performance should be such that it causes the robot to settle back on the path within a second. This translates to a settling time, $t_s = 1s$, for a step or impulse disturbance. In as far as the inner loop controller is concerned, the open-loop system is:

$$G'(s) = \frac{v_x}{s}. \quad (6.9)$$

With a proportional gain of k' , the inner-loop's closed-loop transfer function is:

$$G''(s) = \frac{k'v_x}{s + k'v_x}. \quad (6.10)$$

It has a single pole at the origin and has a root locus as shown in Figure 6.13. Clearly, the effect of the inner-loop controller gain is to move one system pole from the origin towards negative infinity along the real axis as the gain varies from zero to infinity. Our task is to select the value of this gain so that the desired settling time is achieved. The gain was adjusted until it resulted in a settling time of 0.2 seconds – 5 times faster than the desired outer-loop response.

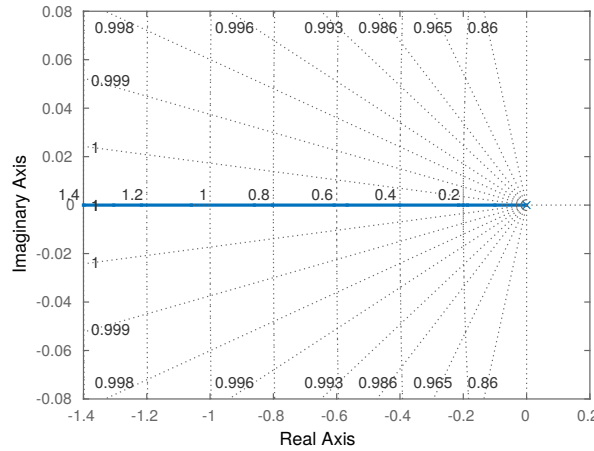


Figure 6.13: Inner loop root locus. The system is stable for all values of gain; higher gain values improve the system's time constant and hence its settling time.

Once the inner loop controller gain is chosen, it is substituted into Equation 6.10 and used for the design of the outer-loop controller. In as far as the outer loop controller is concerned, the open-loop system is the cascaded combination of the inner loop and the final integrator, given by:

$$G'''(s) = \frac{k'v_x}{s(s + k'v_x)}. \quad (6.11)$$

This open-loop system has one pole at the origin and another one on the real axis at $s = -k'v_x$: With a proportional gain of k'' , the outer loop's closed-loop transfer function (which is also the overall closed-loop transfer function of the system) becomes:

$$G_{CL}(s) = \frac{k''k'v_x}{s^2 + k'v_x s + k''k'v_x}. \quad (6.12)$$

On the real axis, the root locus of the closed loop system exists between $s = 0$ and $s = -k'v_x$, with a breakaway point at $\sigma = -0.5k'v_x$ and asymptotes of the root locus approaching positive and negative infinity at angles $\theta_1 = \frac{\pi}{2}$ and $\theta_2 = -\frac{\pi}{2}$ respectively. This root locus is shown in Figure 6.14(a). The system is critically damped for closed-loop pole locations, $s_1 = s_2 = -0.5k'v_x$. The outer-loop control design task is to select an outer-loop gain value that enforces this condition

while maintaining the desired settling time of 1 s. The step response for the overall system is as shown in Figure 6.14(b), clearly having a settling time, $t_s = 1$ s.

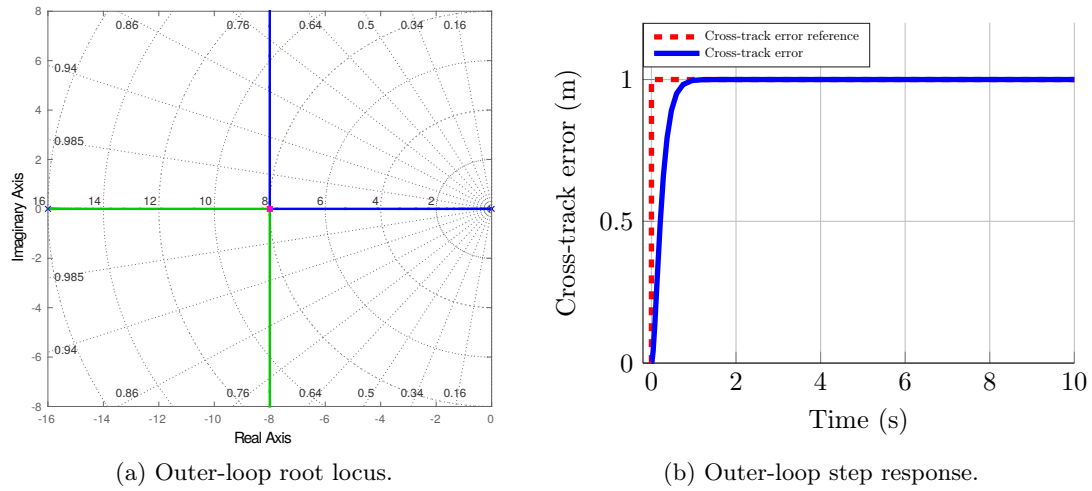


Figure 6.14: The root locus and step response of the outer-loop controller. The system is critically damped, with a settling time, $t_s = 1$ second.

It may seem less intuitive to analyse the system's response to a cross-track error using a step response as this may be easily misinterpreted as commanding the vehicle to deviate from the path. A more intuitive analysis is that of an impulse response, where the system is momentarily disturbed and its behaviour in returning to the nominal path is analysed. As seen in Figure 6.15, the impulse response for the system affirms the result obtained from the step response. As a result, the designed cross-track error controller meets our design specification. This concludes the design of the cross-track error controller. The next section concludes the chapter.

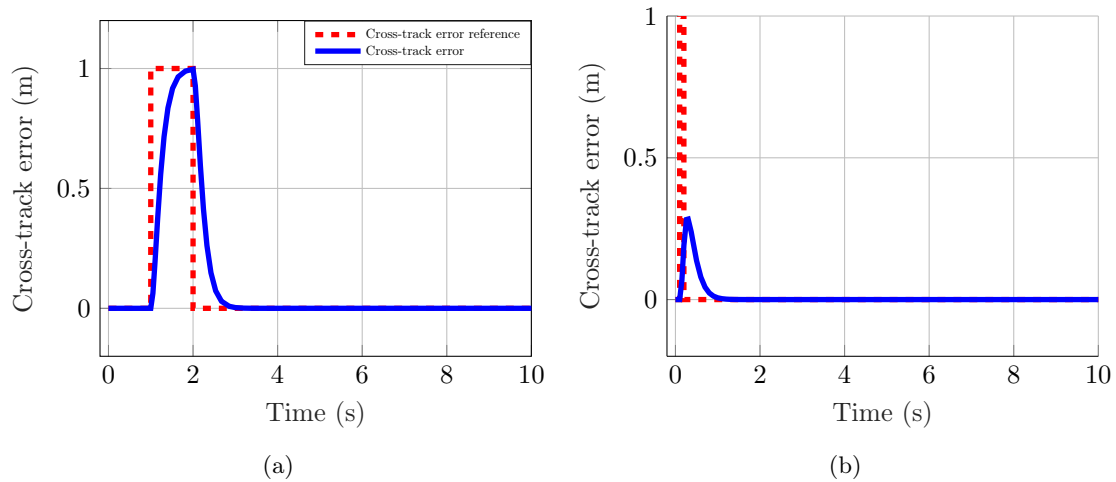


Figure 6.15: An impulse response of the overall system. Sub-figure (a) shows a response to an impulse with pulse duration of 1 second and a magnitude of 1 and sub-figure (b) shows a response to an impulse with pulse duration of 0.1 seconds and a magnitude of 1.

6.3 Path-tracking Controller Summary

This chapter was focused on the design of a controller whose task is to cause the robot to accurately follow planned continuous-curvature paths. The path tracker makes use of feed-forward and feedback control. The feed-forward component uses the curvature profile of the planned path to command the robot to follow the planned path in open-loop fashion. It is noted that feedback control alone is not sufficient, both in the absence of disturbances and when they are present. In the absence of disturbances, due to practical issues, such as discretisation, slight inaccuracies will manifest. In the presence of disturbances the robot will deviate significantly from the planned path with feed-forward control alone. A cross-track error controller has been designed for the purpose of correcting for deviations from the planned path. As a result, the path tracker is now in place. A digitised version of the path tracker has also been implemented in C++. This C++ implementation is used in the deployment of the path tracker through ROS, as part of the integrated autonomous navigation system, which is presented in the next chapter. It is in Section 7.5 of the next chapter where experimental tests of the robot tracking planned paths using the developed path tracker are presented.

Part III

Integrated System Results and Conclusions

Chapter 7

System Integration, Autonomous Navigation Experiments and Results

The previous three chapters presented the development of the different algorithms that are needed to address the optimised path-planning and path-tracking problems tackled by this thesis. Algorithms for computing optimised paths have been developed in Chapters 4 and 5, and the algorithm for following planned paths has been developed in Chapter 6. This chapter presents the integration of these algorithms for the purpose of demonstrating the ability to execute planned paths.

The chapter begins by introducing the mobile robot used to demonstrate path tracking in Section 7.1, touching on both the hardware and software components of this platform. An important aspect of this aspect of this section is the selection of a software framework for integrating the optimised path-planning algorithms with the path tracker and the robot's low-level robot controller to form a system capable of autonomous navigation. We then present a comparison of available simulation environments, with the aim of selecting a suitable one that can be used to virtualise the robot for faster development and enhanced visualisations in Section 7.2. An overview of the integrated system is presented in Section 7.3, outlining how the modules of the integrated system interact. The integrated system is shown to be consistent with the ROS navigation stack, which is a powerful and standard software framework for the deployment of autonomous navigation applications, in Section 7.4. In the same section, we also present an adaptation of the ROS navigation stack to the ESL's AutoNav framework. We then give reasons why the ROS navigation stack was not used in this project, and changes that would need to be effected for the stack to be used in similar projects in the future. Experiments that demonstrate that the integrated system equips the robot with the ability to plan and track optimised paths are presented in Section 7.5. Finally, the work presented in this chapter is summarised in Section 7.6.

7.1 Robot Platform

In this section, we introduce the mobile robot used to demonstrate the execution of the optimised paths planned by the optimised path-planning algorithms developed in this thesis. We outline its physical characteristics, hardware components as well as software options available to interact with it. The discussion on the available software options is particularly important as is it through having a proper handle on this subject that we can effectively deploy our optimised path-planning and path-tracking algorithms on the robot and equip it with autonomous navigation capability.

The robot platform is a Pioneer 3AT (in short, P3AT) skid-steered mobile robot from Adept MobileRobots, herein subsequently referred to as MobileRobots. Figure 7.1 shows an example picture of the robot along with its physical dimensions. Specifications for the robot are shown in Table 7.1.

7.1.1 Robot Hardware and Firmware

The robot comes fully assembled, equipped with batteries, four reversible DC motors (one driving each wheel), motor control electronics, wheel encoders and a 32-bit Renesas P3-SH microcontroller that runs the Advanced Robot Control and Operations Software (ARCOS) firmware. The ARCOS

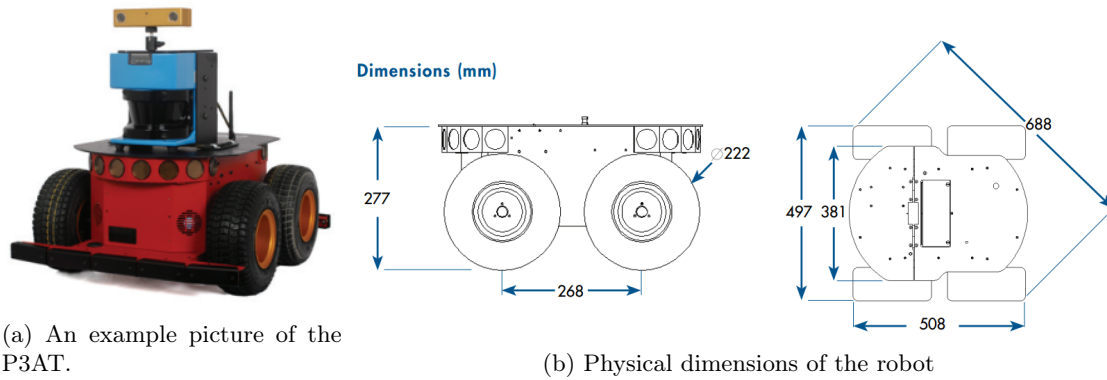


Figure 7.1: Example picture of the P3AT along with its dimensions. Images obtained from the Pioneer 3AT datasheet [33].

Category	Specifications
Skid steer drive	<ul style="list-style-type: none"> • Turning radius: 0 cm; swing radius: 34 cm • Maximum translational speed: 0.7 m/s; maximum rotational speed: 140°/s • Maximum traversable step: 10 cm; maximum traversable gap: 15 cm; maximum traversable grade: 35% • Traversable terrain: asphalt, flooring, sand, and dirt. (low-friction tires available for carpet/indoor use)
Operation	<ul style="list-style-type: none"> • Robot weight: 12 kg • Payload: tile/floor - 12 kg; grass/dirt - 10 kg; asphalt - 5 kg
Onboard computer	<ul style="list-style-type: none"> • 6 X USB2.0 Ports, 2 X PC/104+ Slots, 4 X RS-232 serial ports, ethernet port, wireless ethernet.
Microcontroller IO	<ul style="list-style-type: none"> • System serial, 32 digital inputs, 8 digital outputs, 7 analog inputs, 3 serial expansion ports.

Table 7.1: P3AT specifications

firmware is essentially the robot's operating system. It manages all the robot's peripherals (such as actuators and sensors) as well as its functions (such as acquiring readings from sensors, transmitting motion commands to actuators, and maintaining and distributing the robot's state). Since ARCOS runs on the robot's microcontroller, it is implemented as low-level embedded code. This means that one way for the robot user to modify or extend the robot's functions is to reprogram the flash memory. An interface for uploading code to the microcontroller and debugging is provided for this purpose. This manner of operation is known as the *stand-alone* operation since all the code required for the robot's operations is contained within its microcontroller, without any additional computer. This is illustrated in Figure 7.2.

The stand-alone mode may seem attractive since all code implementing or extending the robot's functionalities is in one place – the robot's microcontroller. However, for robotics applications, such as the optimised path-planning and path-tracking algorithms that have been developed in this thesis, implementation in firmware (i.e. on the microcontroller) is impractical due to processor speed and memory limitations as well as inefficient software development, among other factors. Thus the need to develop these algorithms in high-level programming languages on a powerful piggyback computer that can then be connected to the ARCOS firmware is indisputable. For this reason, MobileRobots has developed a software framework and an application interface (API) to allow for this functionality. This mode of operation is known as the *server mode* and it is illustrated

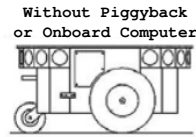


Figure 7.2: Pioneer stand-alone mode. In this case there is no piggyback or onboard computer, hence the only platform to deploy algorithms is the onboard microcontroller, which is a tough call for complex, computationally-demanding robotics applications. Image obtained from the Pioneer 3 operations manual [34].

in Figure 7.3. In this mode, the ARCOS microcontroller serves as a server and client applications are developed through the concepts discussed in the following subsection.

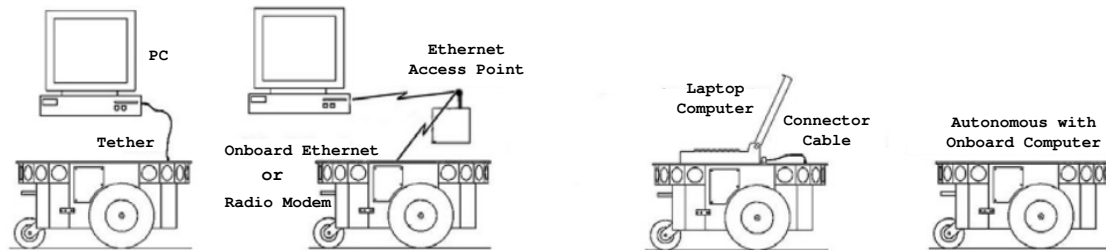


Figure 7.3: Pioneer server mode. Here, two computational platforms are available; the robot's microcontroller for low-level vehicle-specific tasks and a powerful onboard or externally connected computer for the deployment of computationally-demanding robotics algorithms. Images obtained from the Pioneer 3 operations manual [34].

7.1.2 Robot Software

As stated in the preceding subsection, the client-server robot control architecture is more useful than stand-alone mode for the realisation of intelligent robotics applications such as the ones that have been developed in this thesis. Again, the server is the robot's microcontroller, which manages the low-level control details through the ARCOS firmware. For this reason, the robot's microcontroller is also referred to as the *ARCOS server*. It provides an API known as the Advanced Robotics Interface for Applications (ARIA). Through this API, it is possible to develop algorithms for perception, planning and control, using high-level languages including C++, Java, Python and MATLAB, and then deploy them on the robot through the client-server control architecture. This means that it is possible for the optimised path-planning and path-tracking algorithms developed in this thesis to be deployed through this mode. As a result, the ARIA-ARCOS client-server architecture can be used as the software framework for integrating our optimised path-planning algorithms with the path tracker and the robot's low-level motion controller to form a system that is capable of autonomous navigation. When deployed in this manner, our algorithms would run on a computer connected to the robot's microcontroller via a host serial link. This computer can either be an onboard computer that optionally comes with the robot and is directly connected to the microcontroller, or it could be a piggyback computer connected to the microcontroller via a serial cable or via a wireless network as seen in Figure 7.3. This connection would allow our optimised path planner and path tracker to have access to the robot's pose (required by the path tracker) and, a map of the environment (required by the path planner). Once a path has been planned by the optimised path planner, our digitised path tracker can compute motion commands and send them, periodically, via the serial link, causing the robot to follow the planned path. We now present the ARIA architecture to give details about how our algorithms would interact with the robot if the ARIA-ARCOS client-server architecture is chosen as our software framework.

The architecture of ARIA is shown in Figure 7.4. As shown in the figure, at the centre of this architecture is the *ArRobot* class. In any program that is based on ARIA, an object of this class acts

as a communication gateway between that program (ARIA client) and the robot's microcontroller (ARCOS server). This setup abstracts the low-level packet-based communication required by the ARCOS firmware, running on the microcontroller, from the high-level ARIA client. The ARIA client interface (upper part of Figure 7.4) allows developers to write high-level code for specific functions, including connecting to and disconnecting from the robot, performing user callbacks, sending motion commands, commanding the robot to perform specific actions or behaviours, and interacting with the robot's sensors, among others. Of these available interfaces, the applicable ones to our problem are: *connecting and disconnecting from the robot*, and *sending motion commands*. Once connected to the robot, every program based on ARIA gets access to the robot's state, among other information. As such, our path tracker would have access to the robot's pose that it requires for cross-track error computation. This cross-track error is used to compute the feedback component of the corrective angular velocity command generated by the path tracker for the purpose of keeping the robot on the planned path. Once the corrective angular velocity command (feedforward and feedback combined) is computed, it needs to be sent to the robot through the *direct motion commands* interface. Next, we introduce the ArRobot *task cycle*, explaining the processing that happens within it and how our algorithms would fit within it.

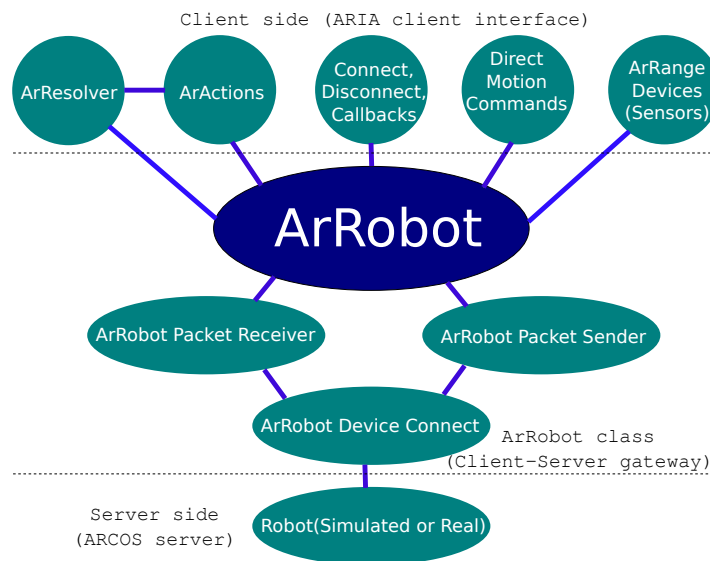


Figure 7.4: ARIA architecture. Adapted from [168].

With a connection established between the robot (simulated or real) and a program based on ARIA, a new server information packet (SIP) is received from the robot at a frequency determined by a user-specified interval known as the task cycle. It is within this task cycle that computations that need to be done based on the new SIP are performed. The end of the task cycle is marked by the sending of commands to the robot. Figure 7.5 shows the processing that occurs within a task cycle. The SIP contains information about the state of the robot, such as the robot's current pose, linear and angular velocities as well as updated sensor readings, among other information – as stated before, for path-tracking purposes, we may be interested in the pose updates for cross-track error computation. The packet is received by the packet handler, whose role is to relay interesting information in the packet to the rest of the computation blocks. First among these other computation blocks are *sensor interpretation tasks*, whose role is to translate sensor observations into a form that is convenient for motion planning and control. As an example, this may involve updating a map that is used by the path planner. Second comes *action and action resolution tasks*, whose role is motion control – this is where our path tracker would be located. Third comes the *state reflection task*, which maintains and distributes a summary of the current operating conditions of the robot, including the latest SIP, the latest sensor readings and cached motion commands. Last comes *user tasks*, which are any other computations that the developer may want to perform in each task cycle, but which cannot be classified as either sensor interpretation or action tasks.

Again, the cycle's end is marked by the sending of motion commands requested by the client to the robot. It is important to note that ARIA provides three ways of controlling the robot, namely *direct commands*, *motion commands* and *actions*. The most basic among these is the use of direct commands, which are made up of a single byte command number followed by arguments (if any) in assembly-like fashion. Direct commands are appropriate for simple tasks like turning on motors. At the second level of the control hierarchy are motion commands, which are simple movement commands that allow, for example, setting the linear or angular velocity of the robot, or stopping the robot, among other motion commands. At the third level of control are actions (also known as behaviours); an action is a sequence of motion commands that causes the robot to move in a certain way, such as go-to-goal, where the robot can move towards a specified position, or avoid-obstacles, where the robot can avoid obstacles detected through sensors by turning. In this project, the path tracker produces velocity commands that causes the robot to follow the planned path. As such, the appropriate level of control for our application is that of motion commands.

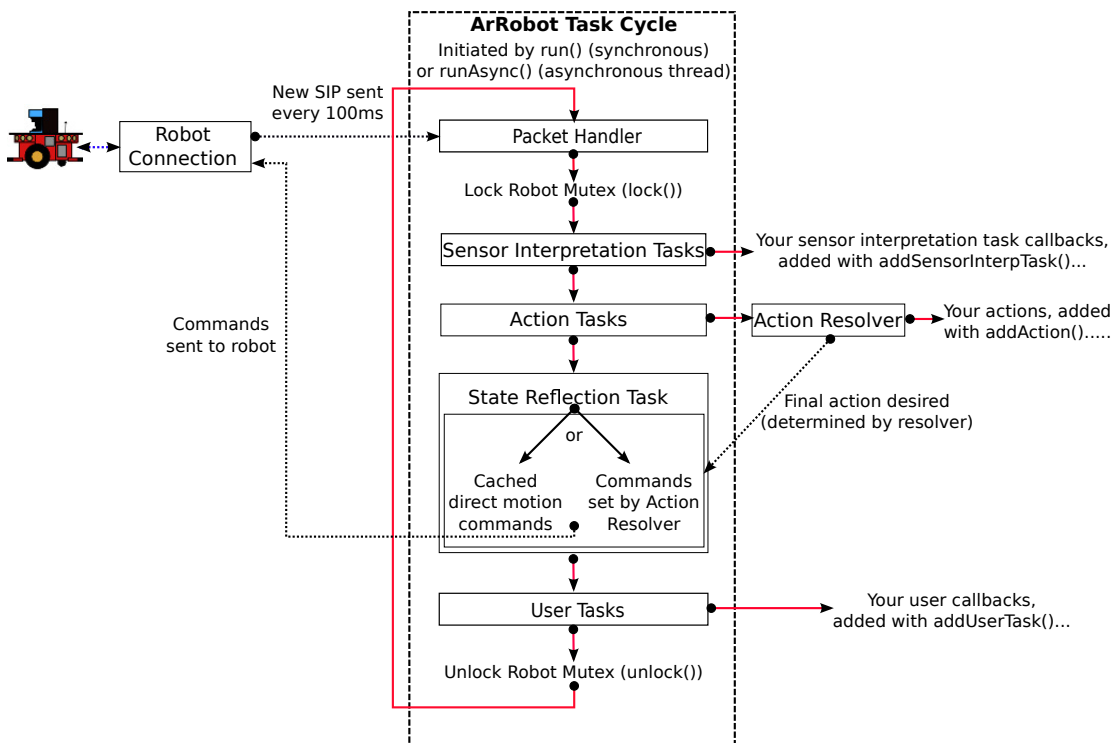


Figure 7.5: Overview of the ArRobot task cycle, adapted from [168].

At this point, the applicability of ARCOS-ARIA client-server architecture as the software framework for deploying our optimised path-planning algorithms and the path tracker has been outlined. While the ARIA API allows for the development of robotics algorithms from scratch, just like we developed our optimised path planners and the path tracker from scratch, MobileRobots also provides a number of off-the-shelf applications based on ARIA for localisation, mapping and navigation, allowing roboticists to use these provided solutions and only develop custom solutions when the provided functionality is not sufficient for the task at hand. We also considered the use of relevant off-the-shelf solutions for this project, but found them to be limiting, as will be explained shortly. Before we get to that explanation, we briefly introduce such off-the-shelf solutions that are applicable to path planning and path tracking:

ARNL (Advanced Robotics Navigation and Localization): It is a collection of software packages, based on ARIA, that have been developed by MobileRobots for localisation and navigation.

Mapper3: It is an application availed by MobileRobots for processing and editing maps for use with ARNL. It is capable of creating 2D maps from laser scan data. It also allows for the creation of maps by adding geometric objects like lines, polygons, and goals, among others.

MobileEyes: It is a graphical application that allows its user to remotely monitor and control a robot. The robot's position as well as other other information pertaining to the operation of the robot, such as range sensor data can be viewed in MobileEyes.

In the context of this project, the off-the-shelf applications listed above can be used as follows: maps created through the Mapper3 application can be used to plan and optimise paths for the robot to follow while ARNL can be used to implement the path-tracking controller and Mobile Eyes used to monitor or visualise the overall navigation mission. While this seems like an attractive route, it is limiting since it is too specific to the particular robot platform. This is because all these tools cannot be used with any other robot platforms except those from MobileRobots. We thus consider the use the power of ARIA in conjunction with the Robot Operating System (ROS), a framework that has been proven in the robotics community to extend well to different robots while also providing a host of tools that are more powerful than the above-listed counterpart solutions from MobileRobots. The versatility of using ROS for this purpose as well as how to use it is discussed in the remaining sections of this chapter.

7.1.2.1 The Robot Operating System (ROS) and ROSARIA

The ARIA library introduced above is very useful for writing stand-alone intelligent robotics applications for the robot platform. However, by the nature of a robotic system, it is often desirable to employ a modular approach and divide the robot's software into small stand-alone executables that cooperate to achieve the overall goal, just as is the case with the various modules of the AutoNav framework. This requires communication between multiple processes, which may or may not be located on the same computer. The Robot Operating System (ROS) [7] was designed for this purpose [174]. It is officially defined as an open-source, meta-operating system for robots that provides services that are similar to those provided by an operating system [169]. The provided services include hardware abstraction, low-level device control, implementation of commonly-used functionality, communication between processes through messages, and management of software packages. In addition to these operating-system-like services, ROS also provides software tools and libraries for writing code, building it and running it, possibly across multiple computers.

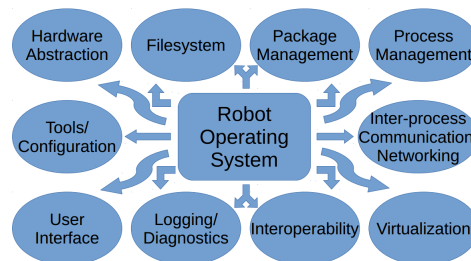


Figure 7.6: ROS provides services that are similar to those provided by an operating system. Sourced from the online reference of ROS [172].

The above definition of ROS is depicted by the functional block diagram in Figure 7.6. In the ROS computation graph, each executable is called a *node*. In the context of the ESL AutoNav framework, we may have a node for each of the mapping, localisation, path planning, path tracking and low-level vehicle control submodules. There are two ways in which ROS nodes communicate. One way is through message passing. This is a broadcast communication method, whereby a node that has information to share (*publisher*) publishes a *message* to a given *topic*. Nodes that are interested in a given topic *subscribe* to it. Once subscribed, subscribers receive all messages sent to the topic, regardless of the publishing node. This way, publishers and subscribers remain

unaware of each other's existence – all they need to know is the *message type* of the topic that they are publishing or subscribed to. A message type is a data structure that forms the basis of a message. It can take any form, ranging from primitive data types to nested data structures. The other way in which ROS nodes communicate is through *services*. Unlike the first, this is a one-to-one communication method. A node that offers a service (*service provider*) advertises it under a *service name*. A node that is interested in the service (*client*) sends a request and awaits a reply. In the case of services, there are two message structures: one for the request and another for the reply. These communication methods are shown in Figure 7.7, where the bottom leg shows communication using topics while the upper leg shows service-based communication.

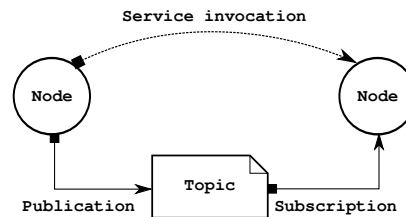


Figure 7.7: Communication between ROS nodes using topics and services. Adapted from the concepts page of the online reference of ROS [172].

Central to the ROS architecture is the *ROS master*, whose role is to manage node registration and lookup for topics, services and parameters in the ROS network – a role similar to that of a DNS server. When initiated, a node reports its registration information to the ROS master. This information includes the topics to which the node publishes and those to which it subscribes, as well as services that it offers and those that it is interested in. As it registers with the master, the node receives information about existing nodes, the topics they publish and services they offer; as such, it can make connections with appropriate publishers and service providers. The master also makes callbacks to all registered nodes when there are changes in registration information (for example, when a new node is registered or when an existing node quits). This allows ROS nodes to create and drop connections dynamically as changes in the network occur. The master also maintains a *parameter server*, which is used to store parameters that are common between nodes. The overall communication process, including the ROS master is shown in Figure 7.8 for topic-based communication. We briefly discuss this communication process in the following paragraph.

The communication process is divided into four phases. Firstly, the nodes (publisher and subscriber) register with the master, specifying their topic names – in this case *scan* (steps 1 and 2). Secondly, the master informs the subscriber about the existence of a publisher of the topic (step 3). The subscriber then initiates a connection negotiation with the publisher (steps 4 and 5). Lastly, with a successful connection negotiation, the connection is made and actual message exchange between the two nodes begins (steps 6 and 7). Henceforth, the subscriber keeps receiving messages from the publisher until one of the nodes quits or dies.

Similarly, service-based communication proceeds as follows: firstly, the nodes (service provider and client) register with the master. Secondly, the master informs the client about the existence of a service provider offering the service. Thirdly, communication negotiation occurs between the service provider and the client. Finally, the actual communication starts, with the client sending request messages and the service provider responding with reply messages.

Through this flexible communication infrastructure, ROS provides a flexible means for decoupling the implementation of a complex robotic system into a number of smaller chunks or modules, each implemented as a ROS node and then coupled at runtime. This is a desirable capability for most robotics applications. In line with this fact, MobileRobots has developed a ROS package known as ROSARIA, which provides a ROS interface to the ARIA API, allowing the use of the power of the ROS architecture for all MobileRobot robot platforms.

As an example, in this project, the optimised path planner, path tracker and low-level motion controllers are decoupled. The robot comes equipped with low-level motion controllers that allow

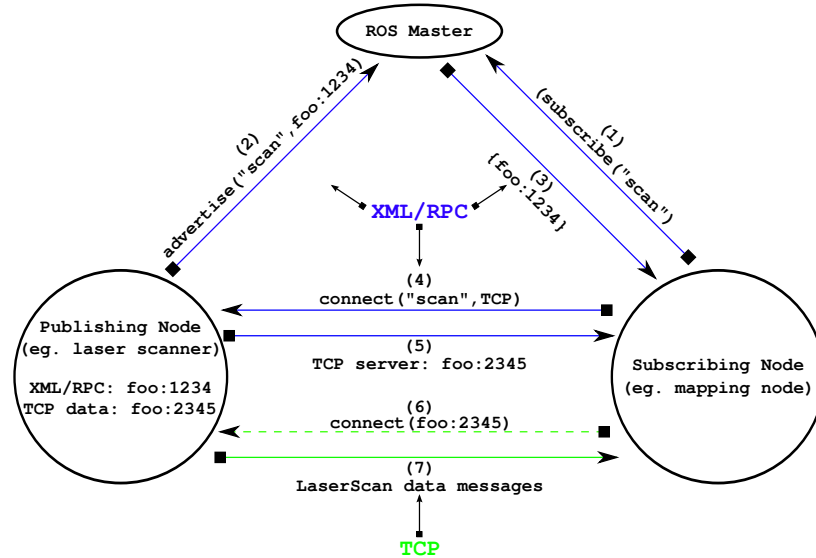


Figure 7.8: Communication between ROS nodes using topics and services with the ROS master. Adapted from the online ROS technical overview [170].

us to command the robot's linear and angular velocities. The optimised path planner and the path tracker have been implemented independently. It is for this reason that it is necessary to have communication between these interacting subsystems. The path tracker needs access to the paths generated by the optimised path planner and the corrective motion commands generated by the path tracker have to be relayed to the low-level motion controllers for execution. The flexible communication infrastructure that ROS provides will be used to facilitate linking of these decoupled subsystems.

Another powerful attribute of ROS, which we introduce in the following section, is hardware abstraction as stated in the official definition. This allows for a seamless change between a physical robot and a simulator since ROS provides the capability to use the same communication interface for both the real robot and a simulator. This means that the developed software does not need to know whether it is communicating to the real robot or the simulator. This enables rapid implementation and testing.

7.2 The Simulated Robot Platform

The previous section introduced the mobile robot used for the demonstration of the ability to track planned paths, most importantly presenting available software options for deploying our optimised path-planning and path-tracking algorithms on the robot platform. The Robot Operating System (ROS) has been presented as a suitable software framework for the integration of our algorithms, which have been implemented in a decoupled manner. It allows us to couple the algorithms at runtime. The hardware abstraction capability of ROS allows for the seamless application of our developed algorithms to either the real robot or a simulator. The use of a simulator helps speed up the development process. Moreover, simulation environments offer 2D/3D visualisations, allowing concepts to be more easily verified, communicated and understood. For this reason, a simulation environment was selected for development and experimentation with the algorithms developed in this thesis. This section presents the selection of a simulator for that purpose. We begin by outlining desirable features for a simulation environment in Section 7.2.1. We then present the considered simulation environments in Section 7.2.2, evaluating each of them against the desirable features outlined in Section 7.2.1. After this evaluation, the simulator that satisfies most of our requirements is chosen.

7.2.1 Desirable Features of a Simulation Environment

This subsection outlines factors that have been taken into account when selecting a suitable simulation environment among existing ones. These factors have been mainly adapted from a comparison of robotic simulation environments presented by Nogueira [174] and they are listed, and briefly described below.

Existence of P3AT robot model: This is the most basic requirement. We require the simulation environment that we will use to have a model of the robot that we are using in this project, the P3AT. If the simulator does not have that model, then we may consider an alternative model that closely resembles the P3AT.

World modelling: To demonstrate optimised path planning and path tracking, it is necessary to have a model of the world in which the robot navigates. We evaluate each simulation environment on how easy it is to create this world model.

Robot model modifications: If necessary, the robot model may have to be modified, for example by adding sensors to it. We evaluate each simulation environment on the ease of performing these modifications.

ROS integration and programmatic control: Since ROS has been selected as the software framework for the development of the integrated system, we evaluate each simulation environment on the ease to use ROS with it. Secondly, our developed algorithms will control a robot in the simulation environment. We therefore need to consider the ease of controlling the simulation using high-level programming languages as well as the available programming languages for a given simulation environment.

7.2.2 Considered Simulation Environments

A variety of simulation environments have been considered. Contesting options for the simulator of choice include MobileSim, V-REP and the Gazebo Simulator. They are briefly introduced below.

7.2.2.1 MobileSim

MobileSim [175] is a proprietary software tool provided by MobileRobots for simulating their robot platforms, the P3AT included, and for experimenting with and debugging applications created using ARIA for these robots. The simulator therefore passes our first requirement, which pertains to the existence of a model of the P3AT. Our second requirement is about the ease of creating a model of the environment in the simulation environment. To model the robot's environment, MobileSim makes use of a file of extension *.map*, known as a map file. This map file can be generated by driving the robot in the environment, scanning it using a laser rangefinder together with the proprietary tools, ARNL and Mapper3, discussed in Subsection 7.1.2. Alternatively, one can survey an environment in which the robot operates and create the map manually using the Mapper3 application as described in Subsection 7.1.2. The created map is composed of straight lines and other basic 2D shapes as shown in Figure 7.9. MobileSim therefore also passes the second requirement. With regard to our third requirement, i.e. the ease of modifying the robot, for example to add sensors, MobileSim also meets it. It allows the user to specify additional accessories to be loaded along with the simulated robot. To interact with the simulated P3AT in MobileSim, one can use MobileRobot's ROSARIA package, which was introduced in Subsection 7.1.2.1. Since MobileSim emulates the robot's ARCOS firmware, the interface between ROSARIA and MobileSim is exactly the same as the interface between ROSARIA and the real robot. This makes MobileSim well integrated to ROS. This means that MobileSim also meets our last requirement, ROS integration and programmatic control.

Although MobileSim meets all the requirements, it has not been selected as our simulator of choice yet, pending consideration of the other two simulators. If another among the remaining simulators meets all our requirements, then the tie will be broken by how well each simulator meets the requirements.

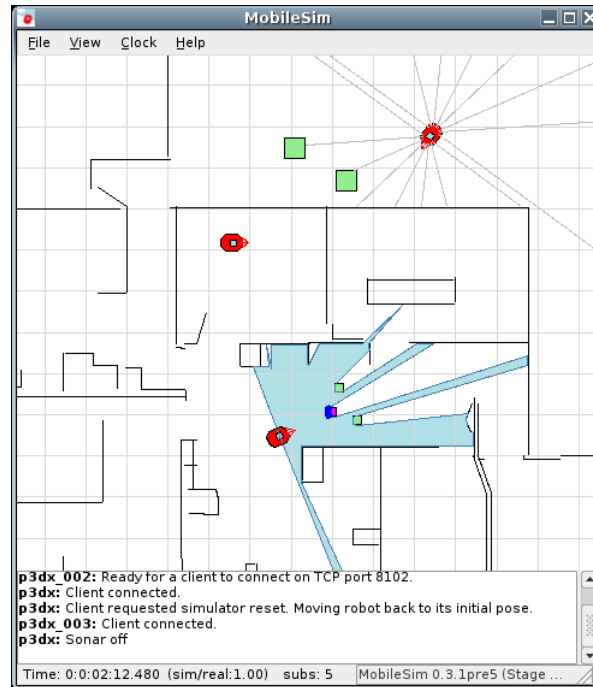


Figure 7.9: MobileSim robot simulator with three P3AT robots. Image obtained by taking a screenshot of a running MobileSim simulation.

7.2.2.2 V-REP

V-REP (short for virtual robot experimentation platform) [176] is a robotic simulator developed by Coppelia Robotics, which is based in Zurich, Switzerland. It is available for both commercial and educational use, with its educational version being available for free. It has support for three operating systems, namely Windows, Linux and Mac. VREP supports seven programming languages; these are: C/C++, Python, Java, Lua, Matlab and Octave. Figure 7.10 shows a snapshot of this simulator, with a sample of objects that can be simulated added in the robot's environment. In its current version, V-REP does not have the P3AT in its model database. An alternative would be to use the P3DX (shown in Figure 7.10), which belongs to the same family of robotic platforms as the P3AT, but instead has three wheels, with one of the wheels being a castor wheel. As such, V-REP does not meet our first requirement of having a model of the P3AT. With regard to our second requirement, i.e. the ease of world modelling, V-REP meets the requirement. It provides a drag-and-drop functionality for adding objects into the robot's environment and a simple interface for editing the properties of objects that are in the environment. In terms of our third requirement, i.e. the ability to modify the robot model if necessary, in V-REP it is possible to add sensors to the robot from inside the simulation environment, without having to edit any files as is the case for example with the Gazebo simulator, which is yet to be discussed. Lastly, with regard to the fourth requirement, ROS integration and programmatic control, V-REP does not have a native ROS node, unlike Gazebo and MobileSim, making it impossible to run VREP as part of a ROS system, but rather alongside it in a separate terminal [178]. For this purpose V-REP offers a ROS plugin that can be used within V-REP scripts to create ROS publishers and subscribers. There also exists a community-developed package [179], which attempts to replicate the features of the Gazebo-ROS plugins package for V-REP. However, due to the lack of a large developer community interested in V-REP, this package lags far behind the former. As such, V-REP is not as well integrated to ROS as MobileSim and Gazebo.

Again, a decision of choice has not been made yet pending the discussion of the Gazebo simulator, which is the subject of the next section. However, at this point it is clear that V-REP meets less of our requirements than MobileSim. Due to this reason, the decision on the simulator of choice boils down to MobileSim and Gazebo. This decision will be made at the end of the

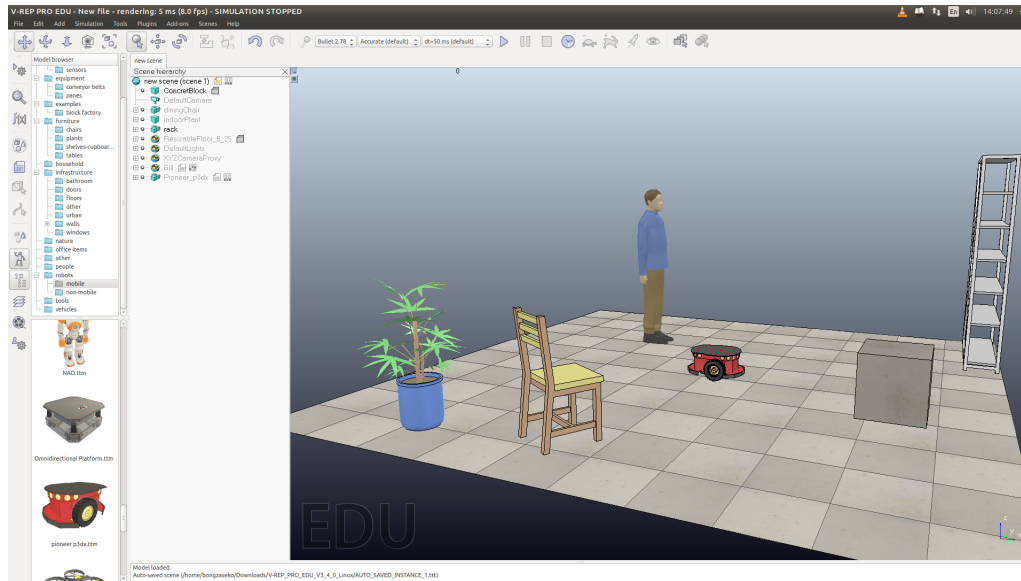


Figure 7.10: The V-REP simulator with a single P3DX robot (the P3AT is not yet in VREP model database). The image has been obtained by taking a screenshot of a running V-REP simulation.

discussion of the Gazebo simulator.

7.2.2.3 Gazebo

Gazebo [8] was invented at the University of Southern California. It was then integrated into ROS framework by a team led by John Hsu at Willow Garage, the original maintainer of ROS. Thenceforth, Open Source Robotics Foundation became the maintainer of Gazebo. Gazebo is an open-source robotic simulator, and is freely available under the Apache 2.0 license. Presently, it is only available on Linux, but support for Windows is planned for future versions. A sample Gazebo simulation is shown in Figure 7.11 where a P3AT, with a laser mounted on it, is simulated in an environment cluttered with obstacles.

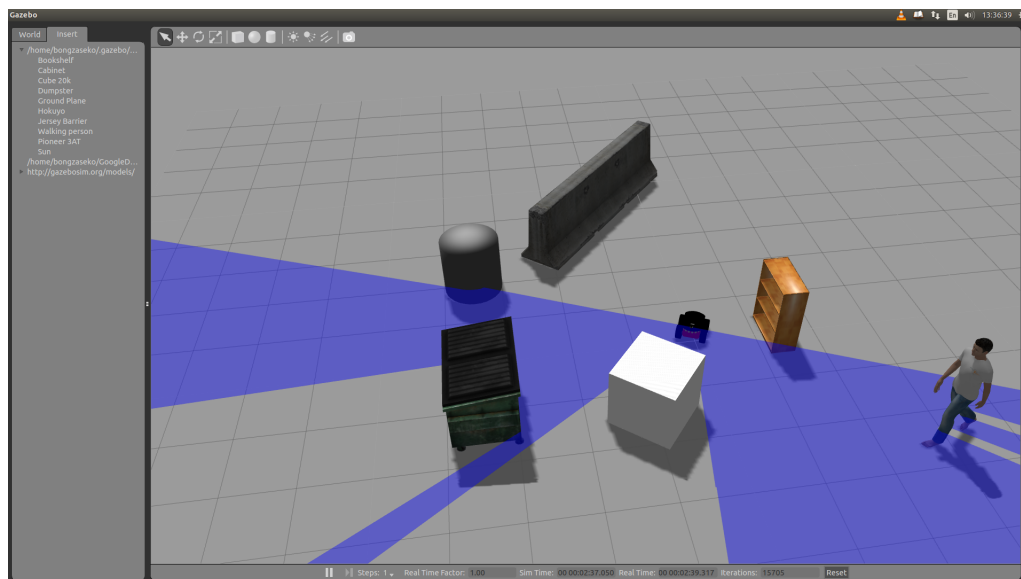


Figure 7.11: The Gazebo Simulator with a single P3AT robot. The image was obtained by taking a screenshot of a running Gazebo simulation.

Our first consideration in evaluating the simulation environment is on whether it has a model of the P3AT. As demonstrated by the example simulation shown in Figure 7.11, Gazebo has this robot model. Moreover, MobileRobots maintains a Gazebo-ROS repository with all their robot platforms, including the P3AT. They also recommend the use of the Gazebo simulator for a more fully-featured simulation [177], giving preference to Gazebo over their proprietary simulator, MobileSim. Secondly, in evaluating the simulator, we consider if it is possible to modify the robot model so as to add robot accessories such as sensors. In Gazebo, although sensor and actuator models as well as plugins for interfacing them to ROS are readily available, there is no way for incorporating them onto the robot from inside the simulation environment. Instead, one has to edit the appropriate *SDF* files and this process has a steep learning curve. Going through this learning experience is, however, worthwhile because it enables one to experiment with complex models. Concerning our third requirement, i.e. the ability to create and edit the world model, Gazebo offers a drag-and-drop capability for inserting existing models of objects into the world; however, to edit them, one has to first learn the SDF format. Changing the objects then constitutes editing SDF files outside the simulation environment. Again, it takes going through a learning experience to master this skill, but once mastered, one can create new models, only limited by one's imagination. Our last consideration pertains to how well the simulation environment is integrated to ROS. Gazebo is the default simulator in the ROS framework [174] and for each ROS distribution, there is a package containing plugins for interfacing ROS and Gazebo in the official ROS repository. These plugins can be attached to objects in the simulated world, enabling the ease to programmatically interact with the simulation through ROS topics and services. One such plugin in the context of the particular platform used in this thesis is the skid-steer drive controller, which subscribes to the *velocity commands* ROS topic and converts the velocity commands to actuator commands, which are then sent to the simulated robot. Other useful plugins include those for a laser scanner, GPS, and IMU among others. This excellent integration of ROS and Gazebo can be seen in Figure 7.12. With both Gazebo and ROS being open source and used widely in the robotics research community, it has more community-developed plugins than any other simulation software, making it to indisputably stand out in this category.

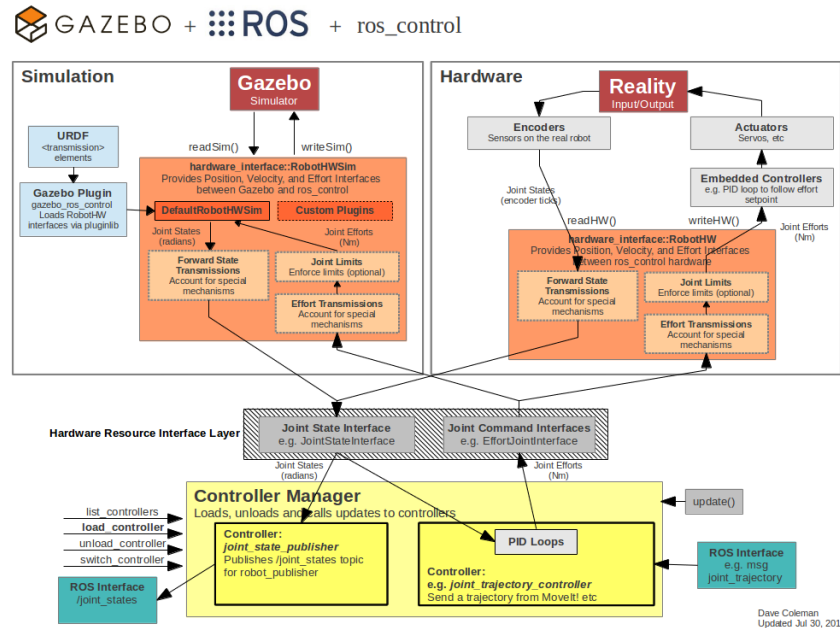


Figure 7.12: Gazebo and ROS integration. ROS uses the same interface for both the simulator and the real robot, allowing a seamless change between the robot and the simulated environment. Sourced from the ROS control page of the online reference of ROS [171].

At this point, we have presented the evaluation of each considered simulation environment

against the desirable features of a suitable simulation environment that were outlined in Section 7.2.1. The next subsection concludes our evaluation of available simulation environments by selecting the one that is most suitable.

7.2.2.4 The Simulator of Choice

As stated in our concluding remarks in Section 7.2.2.2, our decision on the simulation environment to use is between MobileSim and Gazebo. From the comparison, we see that since MobileSim is developed and supplied by the robot's manufacturer, it satisfies most of our desirable attributes of a simulation environment of choice, except for the case of world modelling, where it is unable to include realistic objects into the simulation – only including basic 2D shapes. Gazebo also meets all our requirements, but unlike MobileSim it has a realistic simulation as demonstrated by the visual difference between Figure 7.9, the example MobileSim simulation, and Figure 7.11, the example Gazebo simulation. This is in line with the sentiment from MobileRobots that for a fully-featured simulation, the Gazebo simulator is recommended. Besides, using the Gazebo simulator adds more value to the learning experience since, unlike MobileSim, Gazebo is not specific to robots produced by MobileRobots, but it is used for lots of other robots. As such, mastering this simulation environment does not only add value for this particular project, but also generally to one's career in robotics. Finally, as mentioned in the preceding section, Gazebo is the default simulator in the ROS framework. As such, using it will enable us to fully exploit ROS capabilities. For these reasons, Gazebo has been selected as our simulator of choice.

At this point, the robot that is used in this project has been introduced and the software framework for deploying the developed system has been selected. The simulator to be used for demonstrating the integrated system has also been chosen. The next section presents the architecture of the integrated system.

7.3 Overview of the Integrated System

We now present the architecture of the integration of our optimised path planner and path tracker, and the robot's low-level motion controller into a system that equips the mobile robot with the ability to plan optimised paths and execute them. The system architecture is shown in Figure 7.13. It is as proposed in Section 1.4, with the difference that we are now specific on the communication links between the decoupled subsystems, i.e. the optimised path planner, the path tracker and the robot's low-level motion controller. Particularly, ROS topics are used for these communication links. We also do not show the different components of the optimised path-planning algorithm as we did in Section 1.4. We do so because by now, the optimised path planner has been developed and at this point, it is unnecessary to explain its inner working. In other words, all that matters in system integration are the inputs and outputs of each subsystem.

The optimised path-planning algorithm subscribes to the *robot-pose topic* to which the simulator publishes estimates of the robot's pose. The other two inputs that the optimised path-planning algorithm requires are the path-planning query and the map of the environment. Since, as stated in Section 1.4, the mapping function is outside of the scope of this project, the map is created manually and loaded beforehand. The path-planning query is simply an initial-goal configuration pair. After generating a path, the optimised path-planning algorithm publishes it to the *planned-path topic*. To track the planned path, the path-tracking algorithm requires the planned path and an estimate of the robot's pose. It acquires these inputs by subscribing to two ROS topics, namely the planned-path topic published by the optimised path planner and the robot-pose topic published by the Gazebo simulator. After using this information to compute corrective motion commands that will cause the robot to follow the planned path, the path-tracking algorithm publishes these corrective motion commands to the *motion-commands topic*, to which the robot's low-level controller subscribes. The robot's low-level motion controller then computes electrical signals that are sent to the robot's actuators to cause the robot to move in such a way that it tracks the planned path. The result is a system capable of planning optimised paths and executing them. The integrated system is tested in Section 7.5. Before moving on to testing the system, we first describe a standard ROS framework used for deploying autonomous navigation systems. This is the *ROS navigation stack*. We show how our integrated system can be deployed through it and

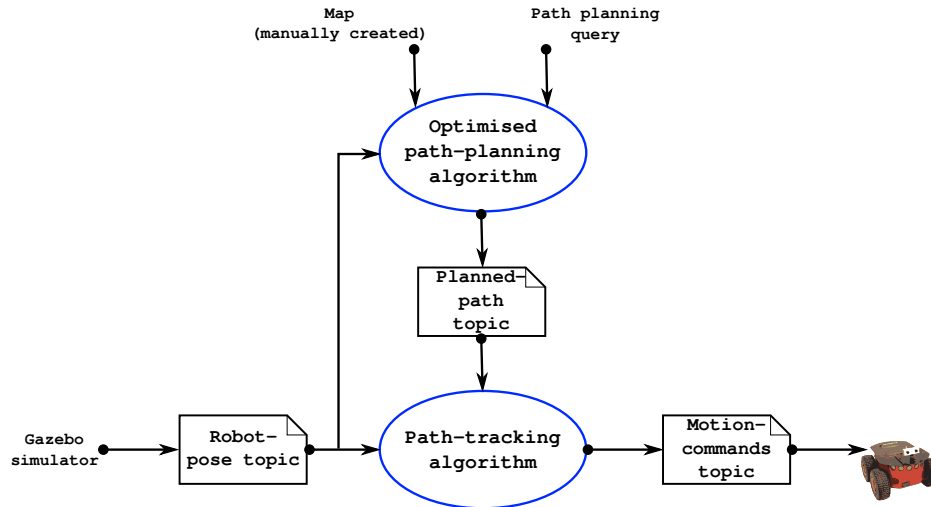


Figure 7.13: The architecture of the integrated system showing the constituting subsystems, their inputs and outputs as well as their interaction through ROS topics.

that the presented system architecture is consistent with the ROS navigation stack. This is the subject of the next section.

7.4 Deployment of the Integrated System through the ROS Navigation Stack

In the previous section we presented and explained the architecture of our integrated system. In this section, we introduce the ROS navigation stack, which is arguably one of the most powerful features of ROS [180]. We show how it can be viewed in the same light the ESL AutoNav framework – the framework on which the system developed in this thesis is based. With the analogy between the ROS navigation stack and the ESL AutoNav framework shown, we then show that the architecture of the system presented in the previous section is consistent with the ROS navigation stack, pointing out how various components of the stack would need to be configured for the deployment of our system through it.

The ROS navigation stack, shown in Figure 7.14, is a ROS package that is designed to move a robot from an initial configuration to a goal configuration without experiencing any collisions. It consists of implementations of a number of algorithms that work together to enable a robot to autonomously navigate its environment. The book authored by Martinez and Fernández [180] contains material on the navigation stack, setting it up and working with it. We introduce the ROS navigation stack by likening it to the ESL AutoNav framework, which we are now familiar with and which serves the same purpose as the ROS navigation stack. We then give reasons why the navigation stack was not used to integrate the subsystems of our optimised path planning and path tracking system.

As shown in Figure 7.14, the stack can be considered as a set of interconnected algorithms or ROS nodes that use sensors of the robot and the odometry to move the robot autonomously using the map of the environment, the robot's odometry and the desired goal configuration. The goal is acquired by listening to the *move_base_simple/goal* topic to which the *global_planner* node subscribes. The map is obtained in a similar way by the *global_costmap* node from the */map* topic published by the *map_server* node. The *global_costmap* node passes the map to other nodes that require it, such as the *global_planner* node and the *recovery_behaviours* node. With the goal configuration and map obtained, the *global_planner* node computes a path to the goal and publishes through the *internal* topic to which the *local_planner* node subscribes. The *local_planner* node uses this path together with an odometry input obtained from a topic with topic name *odom* to compute linear and angular velocity commands that will cause the robot to follow the planned path. These velocity commands are published on a ROS topic named *cmd_vel* to which

the robot's low-level motion controller controller, *base_controller*, subscribes. The *base_controller* uses the reference linear and angular velocity commands to generate signals passed to the robot's actuators to cause the robot to move in such a way that it tracks the planned path. Finally, the *recovery_behaviours* node is responsible for reactive collision avoidance. Essentially, with access the map (obtained from the *global_costmap* node) and sensors capable of detecting obstacles in the proximity of the robot, the *recovery_behaviours* node computes a plan to avoid collisions.

With the ROS navigation stack explained, we now link it to the ESL autonomous navigation (AutoNav) framework, that is, we consider how the stack relates to the AutoNav framework and how it can be configured to emulate the functionality of the AutoNav framework. We start by re-introducing the AutoNav framework. The AutoNav framework, shown again in Figure 7.15, constitutes various submodules grouped into three modules, namely perception, planning, and control as shown. We apply this grouping of the submodules of the AutoNav framework to the ROS nodes of the ROS navigation stack as shown in Figure 7.16. The purpose is to highlight the submodules or nodes performing similar functions in the two frameworks. From this grouping, we can then proceed to discuss how a mapping can be realised between the two frameworks, leading to the ability to emulate the AutoNav framework through the ROS navigation stack.

As stated in the introductory chapter, the path-planning and path-tracking process in the AutoNav framework begins with a high level user input such as a desired goal configuration fed into the path planning and path optimisation module. A global path planner within this module uses the desired goal configuration in conjunction with a map of the environment received from the mapping module to compute a feasible collision-free path connecting the initial configuration of the robot to the desired goal configuration. The path-planning task is divided into three components, namely the global path planner, the local path-planning method (LPM) and a collision detector. Loosely speaking, the task of the global path planner is to find a series of intermediary configurations between the robot's initial configuration and a desired goal configuration through which the vehicle has to go to reach the goal. Once it has found these intermediary configurations, it then makes a call to the LPM whose task is to connect each configuration pair with a feasible path which satisfies the robot's motion constraints. Each local path is confirmed collision-free via a call to the collision detector. The result is a feasible collision-free path connecting the robot to its desired goal configuration. If optimality is desired, this path can be passed to a path optimiser before being

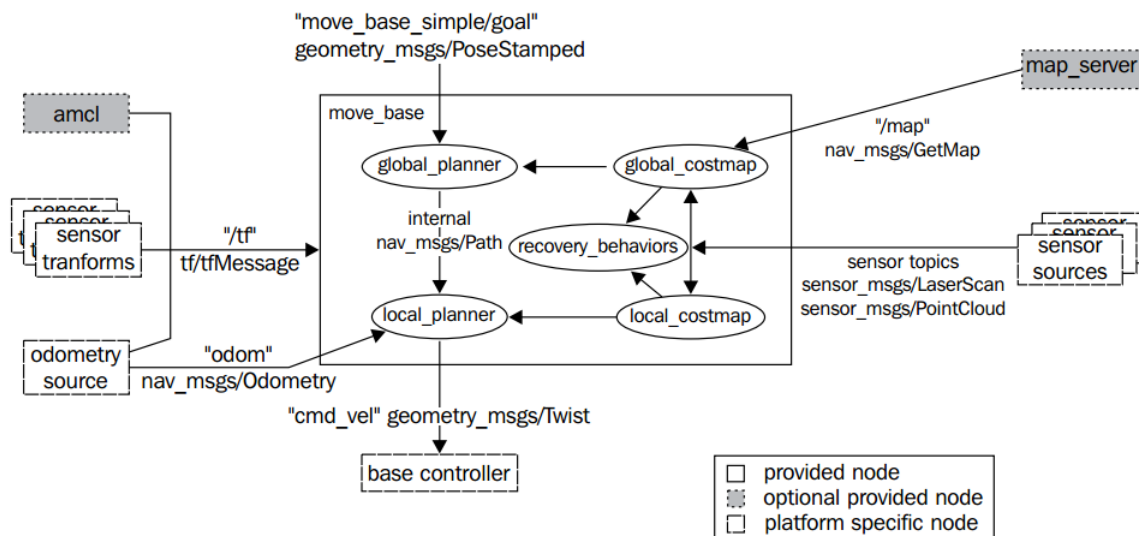


Figure 7.14: The ROS navigation stack can be viewed as a set of interconnected algorithms or ROS nodes that use sensors of the robot and the odometry to move the robot to a goal autonomously configuration without experiencing collisions. The nodes are the rectangular and ellipse blocks while arrows between blocks indicate ROS communication between them. Sourced from the navigation stack setup page of the ROS online reference [173]

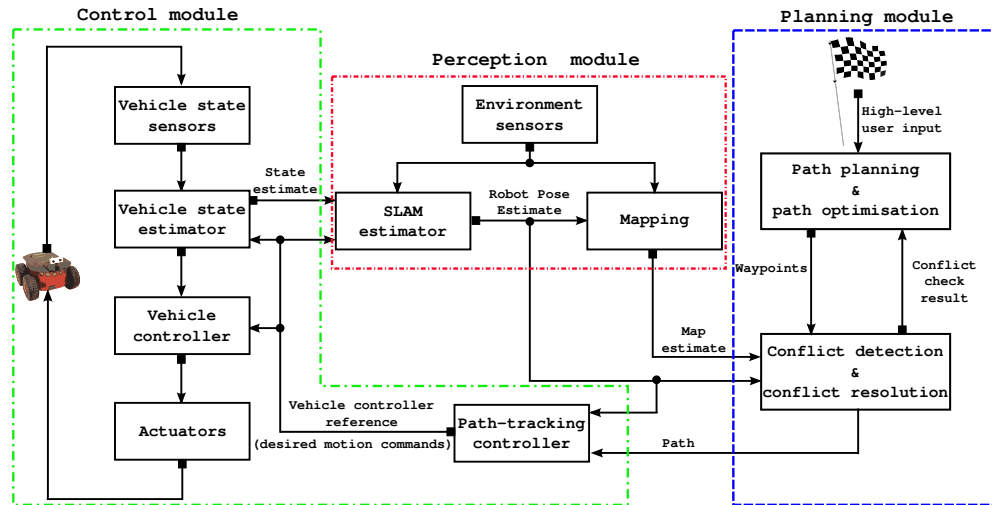


Figure 7.15: The ESL autonomous navigation framework (shown again for convenience).

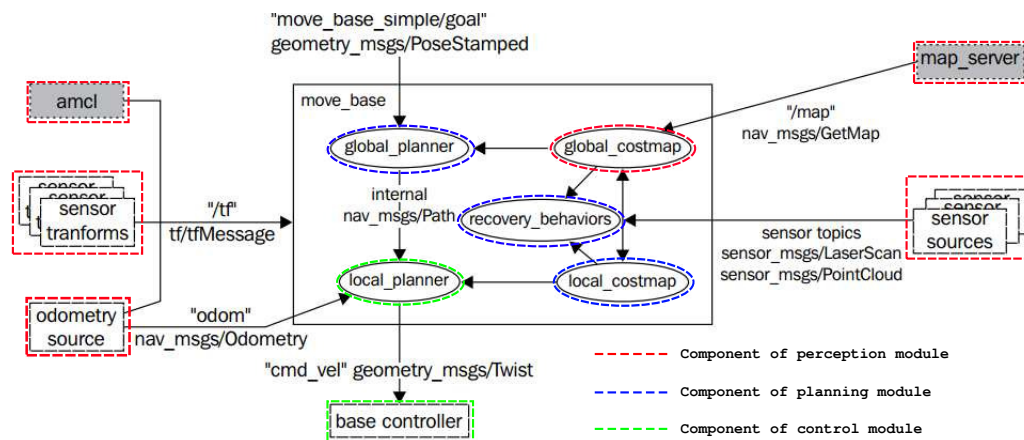


Figure 7.16: The ROS navigation stack in relation to the ESL autonomous navigation framework. The nodes of the stack are categorised according to the modules of the ESL AutoNav framework. Adapted from the navigation stack setup page of the ROS online reference [173].

related to the control module for execution. Additionally, if the robot is operating in an uncertain and/or dynamic environment where portions of the path can be violated by new information, the plan is periodically updated through a process known as replanning and these updates relayed to the control module. The control module uses the resulting path for generating control commands, which cause the robot to track the planned path. Within this module, a path-tracking controller takes the generated path and an estimate of the robot's pose to generate linear and angular velocity commands that when accurately tracked will cause the robot to follow the planned path. The output of the path-tracking controller is then fed to an onboard vehicle controller, which is responsible for the tracking of the commanded linear and angular velocities.

By comparing the description of the path-planning process in the ROS navigation stack to that of the AutoNav framework (both discussed above) we can figure out a mapping between these two frameworks and hence be able to emulate the AutoNav framework through the ROS navigation stack. The result of this mapping process is summarised below.

In relation to the ESL AutoNav framework, it becomes natural to think of the *map_server* as the mapping submodule, while the *amcl* (adaptive Monte Carlo localisation) node represents the localisation module. The conflict detection and conflict resolution submodule in the AutoNav framework can be thought of as living within the *recovery_behaviours* node of the navigation stack, while the path planning and

optimisation submodule is equivalent to the *global_planner* node. The path-tracking controller submodule of the AutoNav framework is equivalent to the *local_planner* node of the stack; this introduces an ambiguity in terminology between the AutoNav framework (which is in line with the global motion planning and control literature) and the ROS navigation stack, as the local path planner submodule of the framework serves an entirely different purpose from that of the *local_planner* node of the stack. However, awareness of this ambiguity is sufficient for one to properly map between the two frameworks. Finally, the *base_controller* node represents the onboard vehicle controller submodule of the framework.

Thus, with respect to the navigation stack, this project can be viewed as aimed at implementing ROS algorithms for the *global_planner* and *local_planner*. This is because these are the nodes that are respectively tasked with planning paths and tracking them. It should be noted that the *recovery_behaviors* node can be viewed as playing a role in path tracking. Particularly, in the ROS navigation stack, the *recovery_behaviors* node is called upon in the event that while tracking the planned path, the robot finds itself encountering an obstacle. This may be due to a number of factors: perhaps a new obstacle may have entered the robot's environment, or it might be that the robot deviated from its plan, or maybe the path planner failed to account for an obstacle when generating the plan¹. Whatever the case may be, if this happens, the robot suspends following of the planned path and calls upon *recovery_behaviors* to get it out of the collision course. Following the path is resumed once the robot is clear of the obstacle. This approach can be thought of as reactive collision avoidance. Reactive collision avoidance is not in the scope of our project; rather, in this project we plan paths that are collision-free and the task of the path tracker is then to follow these paths accurately. As such, with respect to the project, the *recovery_behaviors* node would have to be disabled if the ROS navigation stack is used. The other node that facilitates reactive collision avoidance in the ROS navigation stack is the *local_costmap* node. It basically maintains the portion of the global costmap that lies within a square window, whose window size specified by the user, around the robot. It is this local costmap that is periodically scanned by the *local_planner* node to determine whether path following should be suspended. The *recovery_behaviors* node also uses this map when attempting to get the robot off the collision course. Since we do not make use of reactive collision avoidance in this project, we can also disable the *local_costmap* node.

With the *recovery_behaviors* and *local_costmap* nodes disabled, as shown in Figure 7.17, the ROS navigation stack resembles our system architecture given in Figure 7.13. As such, the integrated system is consistent with the ROS navigation stack, which is a powerful and standard framework for the deployment of autonomous navigation applications.

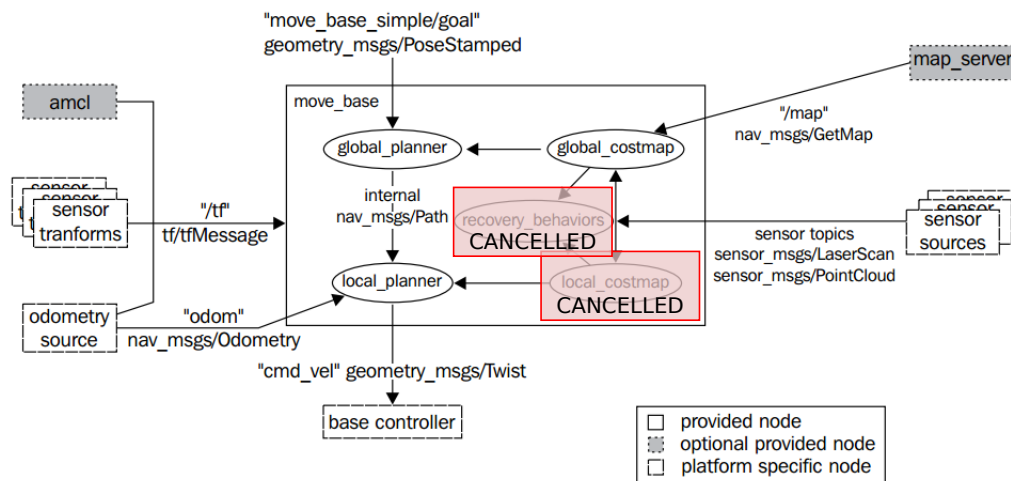


Figure 7.17: With the *recovery_behaviors* and *local_costmap* nodes disabled, the ROS navigation stack resembles our system architecture (Figure 7.13). Adapted from the navigation stack setup page of the ROS online reference [173].

¹This list of possible factors is only meant as an example and it is not exhaustive.

An important aspect of the ROS navigation stack to note is that it uses an occupancy grid map as its map representation. We mentioned in Section 5.1.1 that using occupancy grids in our application would make path planning prohibitively slow since there would be many cells along certain paths, making the collision tests expensive. We also mentioned that an occupancy grid map can be used if there is an efficient way to first to convert it to our selected map representation, i.e. a piecewise-linear map; unfortunately no solution of this nature exists yet. It is also important to reiterate that mapping is outside the scope of the project. For our purpose, it suffices to manually create the map and then pass it to the path planner.

Due to these issues the ROS navigation stack was not used for the deployment of the project. Instead, the optimised path planner and the path tracker were each written as ROS nodes. As explained in Subsection 7.3, the optimised path planner publishes generated plans to the *planned-path topic* to which the path tracker listens. Then to control the robot to track the planned path, the path tracker publishes the generated motion commands to the *motion-commands topic* to which the robot's low-level controller listens.

The next section proceeds to the demonstration of the integrated system whose architecture was presented in the previous section.

7.5 Autonomous Navigation Experiments and Results

Algorithms for solving the optimised path-planning and path-tracking problems have been developed in Chapters 4, 5 and 6. All of the developed algorithms have been implemented in MATLAB and C++. The first two sections of this chapter have detailed how the developed algorithms have been integrated through the use of the Robot Operating System (ROS) to equip a P3AT mobile robot with autonomous navigation capability. This section presents results for experiments conducted using a P3AT, simulated in Gazebo, to demonstrate the ability of the overall system to plan and execute optimised solution paths for given path-planning queries. The reason for only using the simulated robot platform is that at its present state, the physical robot platform is not equipped with a localisation system. As a result, a decision had to be taken on whether it was appropriate to dedicate time to put a localisation system in place, thereby reducing the amount of time available for the development and analyses of the optimised path-planning and path-tracking algorithms, which are the focus areas of the project. The decision was that time to put the localisation system in place might be too much, and would infringe the actual focus areas, given that the algorithms also had to be ported to C++. As a result, we resolved that the system would instead be deployed on the simulated robot, with virtual disturbance that represents disturbances that can be encountered by the actual robot in the real world artificially imposed. Two options have been considered for imposing the virtual disturbance. One is by adding noise to the perfect state measurements received from Gazebo and the other is to introduce virtual drift on the robot while it is executing the planned path. This disturbance will then be corrected by our path-tracking controller, demonstrating that it would be able to cause the real robot, for which disturbances are inevitable, to track a planned path. The second option has been elected and used.

Figure 7.18 shows a snapshot of a run of the C++ implementation our path planner, deployed in ROS. This implementation of the path planner allows for the selection of any of the 3 benchmark path planners and the 12 optimised path planners. In Figure 7.18, obstacles are represented by red boxes, while the initial and goal configurations are respectively depicted by the red and green arrows, and the planning tree and the solution path are respectively coloured blue and green. With this path planned, the path tracker then causes the robot to follow it. The rest of this section presents our analyses on how well the path is tracked. The purpose is to show that paths generated by our path planners are executable. We begin by demonstrating the ability of the system to accurately track planned paths, with only slight inaccuracies, in the absence of disturbances (Subsection 7.5.1). Thereafter, in Subsection 7.5.2 we show that the system is capable of returning to accurately tracking the planned path after deviation. While multiple path-planning and path-tracking scenarios may have been presented in our demonstrations, our position is that since any planned path is composed of continuous-curvature path primitives, including clothoids, circular turns and straight lines, then demonstrating the system with a path that contains these primitives is sufficient to demonstrate that the planned paths are indeed executable.

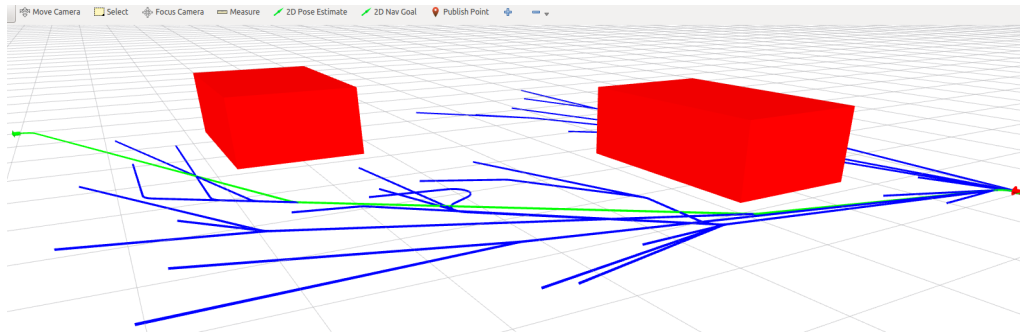


Figure 7.18: The generation of the path planned by our optimised path planner. Image captured from the ROS visualisation tool, RViz. The path is 136.9 m long. Obstacles are the red boxes, the planning tree is coloured blue, and the solution path is green.

7.5.1 Path Tracking in the Absence of Disturbances

In Figure 7.19, we show the 136.9 m-long planned path, with the actual path executed by the robot plotted over it, for the case when the path tracking controller is purely based on feedback control. Figure 7.20 then gives a plot of the cross-track error as a function of time during the execution of the path for this case. The robot starts from rest at the initial configuration and follows the path until it approximately reaches the goal configuration – a cross-track error of 0.037 m is observed at the goal configuration. Along its way to the goal, the robot tracks the path well, with the highest tracking accuracy observed in the tracking of straight line segments – as supported by the cross-track error that is close to zero along straight-line segments. This is in contrast to the tracking of turns wherein the cross-track error is up to 0.05 m. The time periods where the cross-track error is large in Figure 7.20 correspond to just after leaving the initial configuration, as well as at each turn, including the turn just before the goal configuration. These errors occur due to the nature of our waypoint scheduler, as explained shortly. In tracking the path, the role of the waypoint scheduler is to constantly update the current source waypoint. At any point in time, the path tracker tracks the path segment starting from the current source waypoint and ending at the current destination waypoint (the upcoming waypoint). The waypoint scheduler updates an upcoming waypoint to be a source waypoint immediately after that waypoint is reached by the robot. As a result, the robot always overshoots all the waypoints, and the cross-track error controller only gets the chance to correct this error after it has been detected. Next, we consider the case where the path-tracking controller has both the feed-forward and feedback components.

Figure 7.21 shows the planned path, with the actual path executed by the robot plotted over it, for the case when the path-tracking controller is composed of both feed-forward and feedback control. Figure 7.22 then shows a plot of the cross-track error as a function of time during the execution of the path for this case. Again, the path tracker does well to follow the path until it approximately reaches the goal configuration – with the robot stopping within 0.013 m from the goal configuration. Along the robot's journey to the goal, the highest tracking accuracy is again observed for straight-line segments, which again have almost zero cross-track error. However, this time, with feed-forward control added, the tracking accuracy for turns is improved as shown by the decrease in the worst-case cross-track error. Again, the time periods where the cross-track error is large on the cross-track error plot are due to the behaviour of our waypoint scheduler, as explained earlier. The next subsection presents experiments that demonstrate the ability of the path tracker to restore the robot to accurate path tracking after being momentarily disturbed.

7.5.2 Path Tracking in the Presence of Disturbances

The preceding subsection has presented results that show that the integrated system is able to accurately track planned paths in the absence of disturbances. Ultimately, for a real robot operating in the noisy real world, disturbances will be encountered that will cause it to deviate from the planned path while executing it. Such disturbances may, for example, emanate from unevenness of the terrain, wind resistance, imperfect actuators, and unbalanced tire pressures, among other

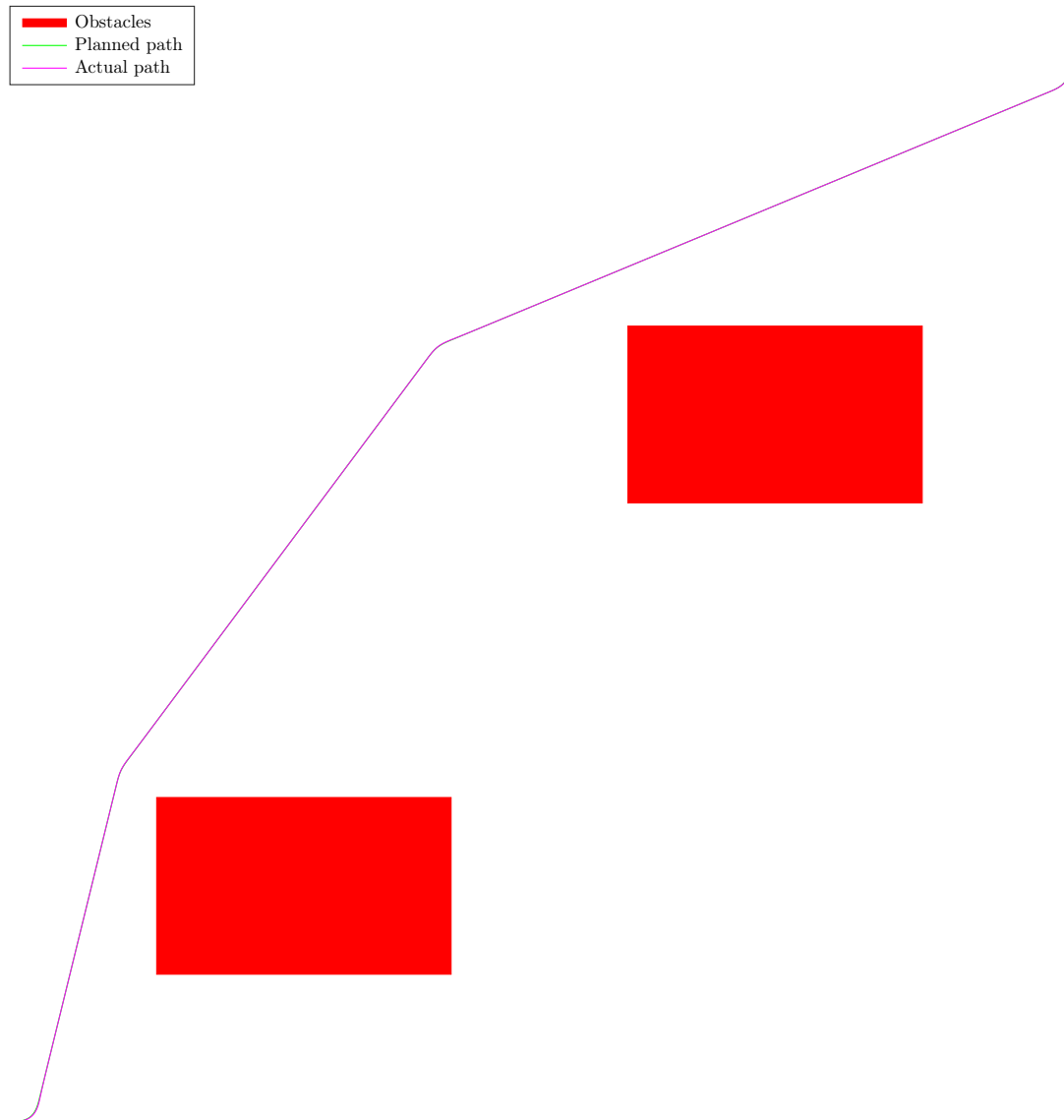


Figure 7.19: A plot of the planned path, generated in Figure 7.18, with the actual path, followed by the robot, overlaid. This is for the case where the path tracker is based on feedback control only, and in the absence of disturbances.

possible sources of disturbance. While in our lab we have the mobile robot used in this thesis, the P3AT, at the present moment, it is not equipped with a localisation module and as a result it has not been possible to use it demonstrate the developed system practically. As a result, to demonstrate path tracking in the presence of disturbances, we still use the Gazebo simulation environment. In its present state, our simulation also does not exhibit the mentioned disturbances, as shown by the results of the previous subsection. We therefore resort to virtually imposing disturbance on the robot while it is tracking the planned path to test our path tracker's ability to cause the robot to return to the planned path after deviation. In imposing this artificial disturbance, we note that, generally, the effect of the disturbance is that it counters the action of the path tracker in controlling the robot to track the planned path by, instead, causing it to drift from the planned path. A strong disturbance will result in a large deviation from the planned path, demanding more control effort from the path tracker. Weaker disturbances will result in slight deviations, which can be easily corrected. Our experiment then constitutes overriding the path tracker momentarily,

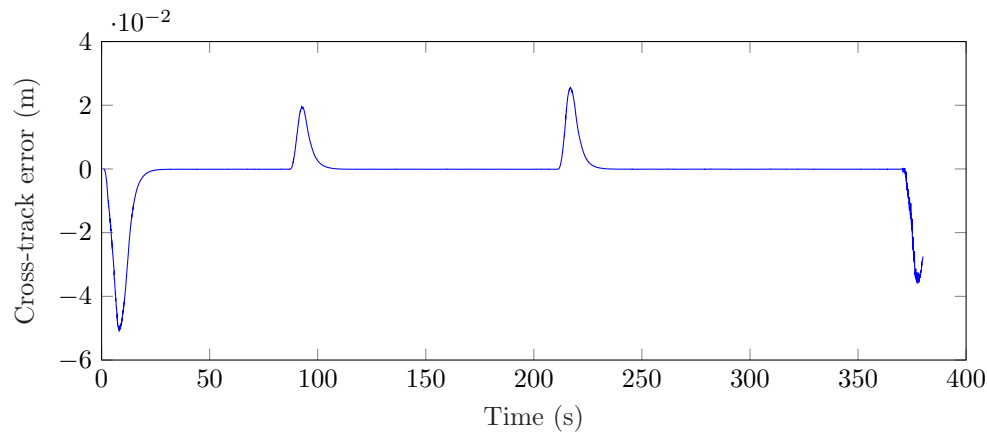


Figure 7.20: A plot of the cross-track error as a function of time, for the case where the path tracker is based on feedback control only, and in the absence of disturbances.

and instead giving a deviation command to the robot. With the robot having deviated by some desired amount, the override is lifted from the path tracker so that it attempts to restore the robot to accurately tracking the planned path. Results from the performed experiment are presented next.

Figure 7.23 shows the planned path, with the actual path executed by the robot plotted over it, for the case when the path-tracking controller is composed of both feed-forward and feedback control. Figure 7.24 then shows a plot of the cross-track error as a function of time during the execution of the path for this case. Seven virtual disturbances were imposed on the robot while it tracked the planned path. Without disturbances, the robot does well, tracking the planned path with only slight inaccuracies, as before. When the virtual disturbances are imposed, the robot deviates from the planned path; however, once the override is lifted from the path tracker, the cross-track error controller is able to restore the robot back to following the planned path. The imposed disturbances resulted in cross-track errors of various magnitudes. The largest one, which happens at a turn, is 1.4 m. This translates to 140% of the maximum cross-track error for which the cross-track error controller was designed. For this large cross-track error, the robot slightly overshoots the planned path on its return, but settles thereafter. The next subsection gives a summary of the experiments performed.

7.5.3 Summary: Autonomous Navigation Experiments and Results

This subsection presented experiments performed to demonstrate the ability of the path tracker to cause the robot to follow the planned path. Two sets of test have been presented. The first set of tests were performed to demonstrate the ability of the path tracker to track the planned path with only slight inaccuracies in the absence of disturbances. On the other hand, the second set of tests were performed to demonstrate the ability of the path tracker to restore the robot back to the planned path after deviation.

The presented results demonstrate the ability of the path-tracking controller, designed in Chapter 6, to track paths generated by the path-planning algorithms developed in Chapter 5. In the absence of disturbances, the system has been shown to be capable of accurately tracking the planned path, resulting only in slight inaccuracies that can be attributed to practical controller implementation issues, such as discretisation and sampling, wherein the control signal is only applied at sampled intervals and held constant within the sampling period. Another observed source of tracking errors is the nature of our waypoint scheduler, which only marks an upcoming waypoint as the new source waypoint after passing it. This results in the robot overshooting all waypoints as the path tracker only begins tracking the new track after its source waypoint has been passed. The second set of results has demonstrated that if the robot encounters disturbances while tracking the planned path, the path tracker is capable of restoring the robot back to accurately tracking the planned path. The overall result is a system that is capable of planning and executing optimised

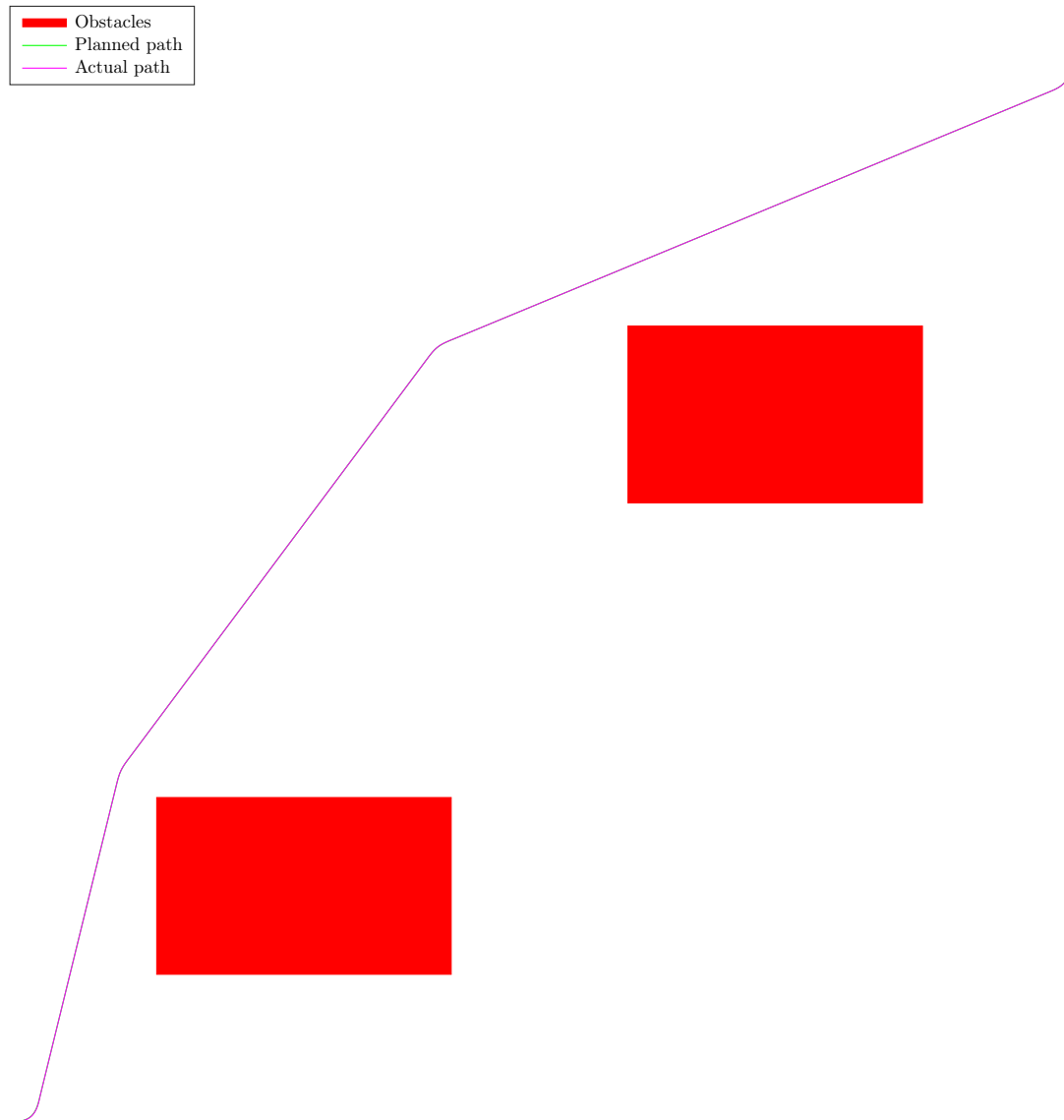


Figure 7.21: A plot of the planned path, generated in Figure 7.18, with the actual path, followed by the robot, overlaid. This is for the case where the path tracker has both feed-forward and feedback control, and in the absence of disturbances.

paths, in line with the objectives of this project. The next section summarises the work presented in this chapter.

7.6 Chapter Summary

In this chapter, we considered the experimental research platform used for this project, examining its hardware, firmware and software. We established the need for a client-server configuration – where the low-level control and housekeeping functions of the robot are handled by an onboard microcontroller, which acts as a server, and the high-level robotics applications, which are the subject of this project, are deployed on a powerful onboard or piggyback computer which acts as a client in the system architecture. We then proceeded to consider the options through which the client applications can be developed. We first considered the powerful ARIA API and went on to motivate the use of the Robot Operating System (ROS) together with it due to the ease of

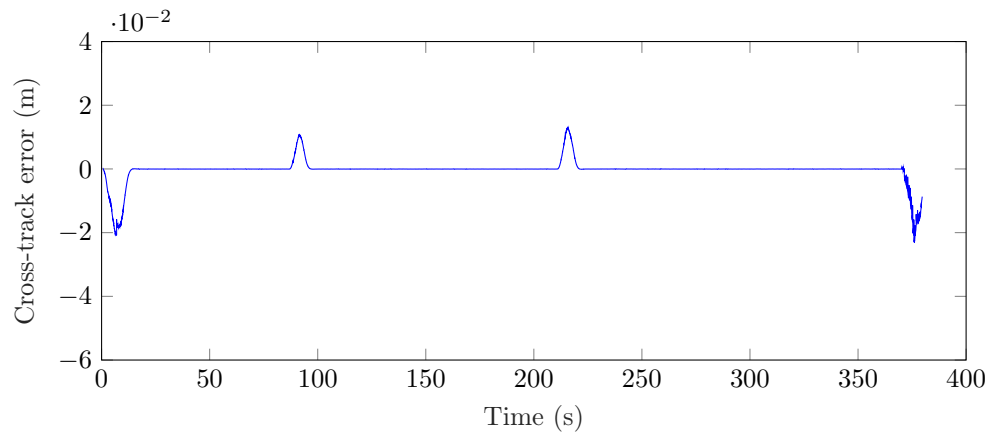


Figure 7.22: A plot of the cross-track error as a function of time, for the case where the path tracker has both feed-forward and feedback control, and in the absence of disturbances.

decoupling the modules of the robotic system at implementation and coupling them at runtime. We pointed out that the use of ROS is an enabler for the use of the popular Gazebo simulator, which is useful for realistic simulation and visualisation of robotics applications. It also enables better extensibility of our solutions to other robot platforms other than those from MobileRobots.

Once the relation between the real robot and simulation was discussed, the focus shifted to figuring out a mapping by which the ROS navigation stack can be used to emulate the functionality of the ESL AutoNav framework, which is the basis for the robotics research in our lab. This prepared the ground for C++ implementations of the optimised path-planning and path-tracking algorithms developed in Chapters 5 and 6 to be deployed in ROS. Finally, the chapter was concluded with autonomous navigation tests, whose purpose was to demonstrate the working of the integrated system. As part of these tests, a simulated Gazebo world in which the autonomous navigation tests were to be performed was created, and the map of the environment was built. The overall system was then demonstrated through the execution of a path planned by the ROS C++ implementation of our optimised path-planning algorithm.

The demonstration of the ability to plan and execute optimised paths marks the end of the thesis. The next chapter summarises the work presented in this thesis, draws conclusions and suggests possible directions for future work.

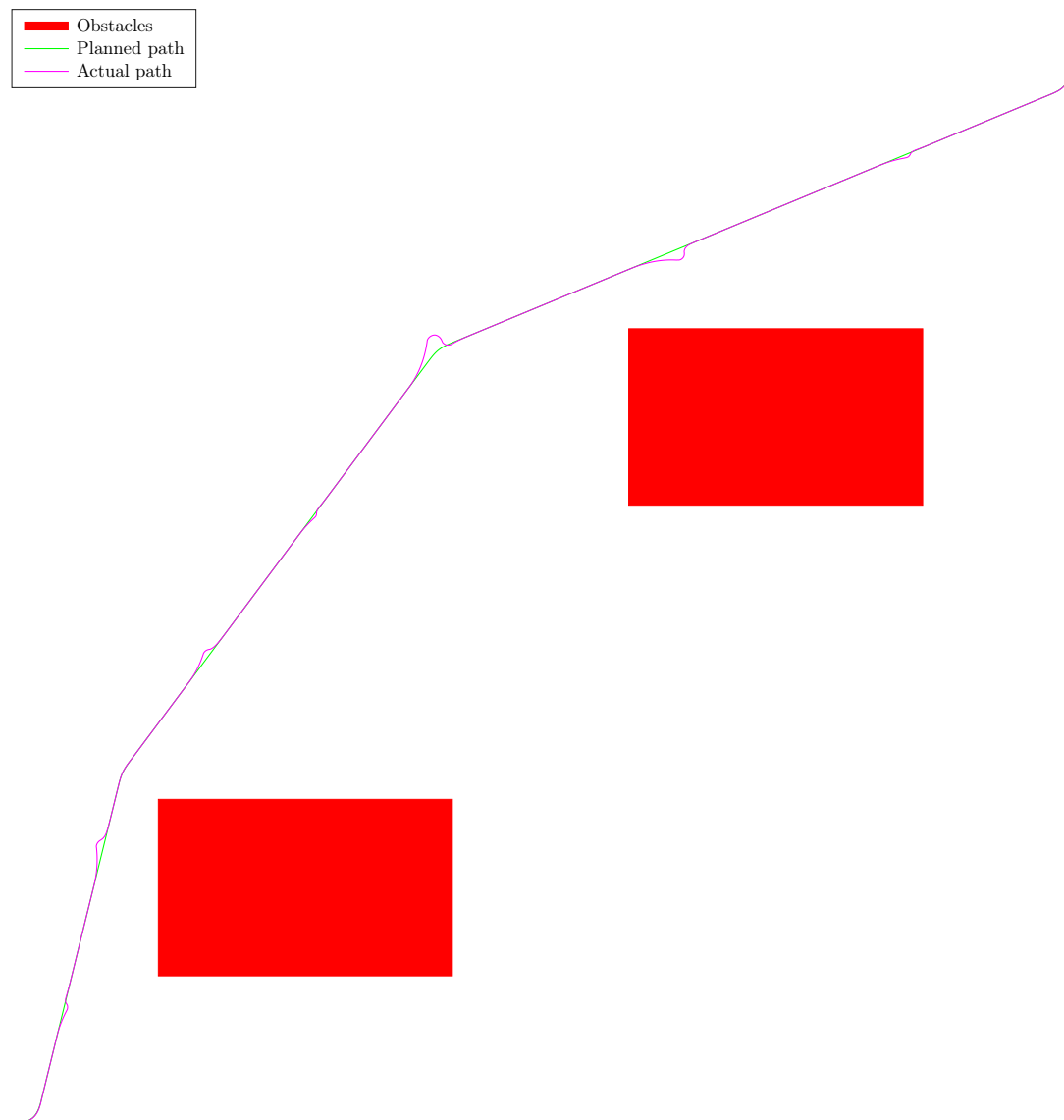


Figure 7.23: A plot of the planned path, generated in Figure 7.18, with the actual path, followed by the robot, overlaid. This is for the case where the path tracker is based on both feedforward and feedback control, and the robot encounters disturbances while tracking the planned path.

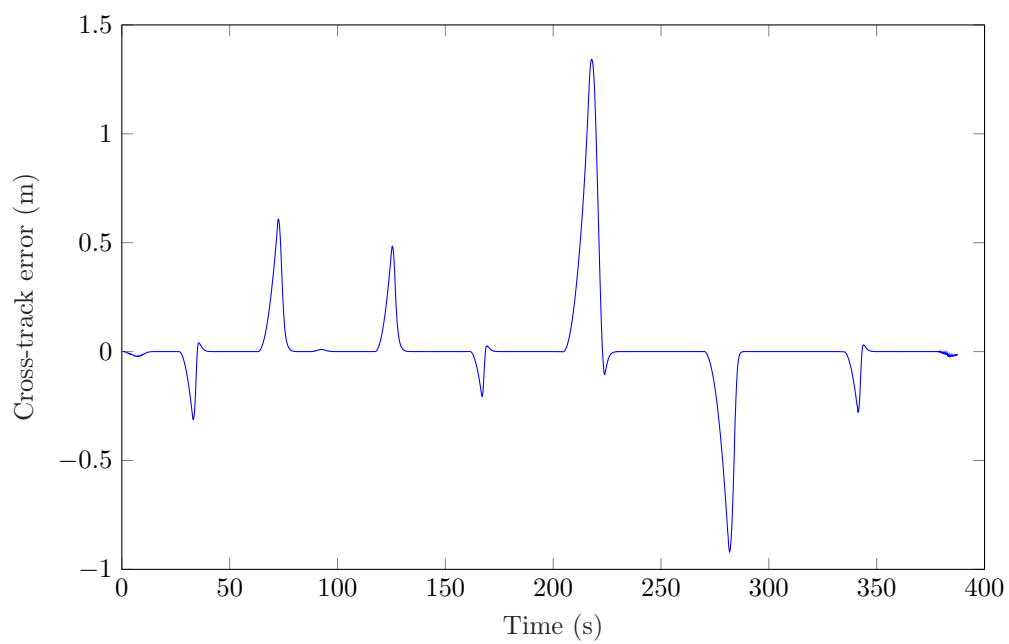


Figure 7.24: A plot of the cross-track error as a function of time, for the case where the path tracker is based on both feed-forward and feedback control, and in the presence of disturbances.

Chapter 8

Conclusion and Future Work

In this closing chapter, we reflect on the work done in this project with the aim of providing a high-level overview and evaluation of what has been done, as well as suggesting directions for future research. Section 8.1 summarises the work done, weighs the achievements of the project against the research objectives outlined in Subsection 1.4.3 to arrive at a conclusion about the contributions of the project. With the conclusion drawn, Section 8.2 wraps up the chapter (and the thesis) by discussing possible directions for future work.

8.1 Summary and Contributions

8.1.1 Summary

This thesis focused on the development of algorithms that equip a mobile robot with the ability to plan and execute collision-free optimised continuous-curvature paths in an environment that contains obstacles.

The quest for optimised path planning and path tracking resulted in the development of algorithms that fall into five groups, namely local path-planning algorithms, benchmark global path-planning algorithms, path-optimisation algorithms, optimised global path-planning algorithms and a path-tracking algorithm. All these algorithms were developed with the requirement of the produced paths being of a continuous-curvature profile in mind. The main focus of the project was on optimised global path planning, which was achieved by combining existing path-planning techniques (i.e. the benchmark global path-planning algorithms) with path-optimisation techniques, for accelerated convergence towards the optimal solution. It is in the development and analyses of these algorithms that most of the time was spent and, ultimately, where the largest contribution was made. The benchmark path-planning, path-optimisation, and optimised global path-planning algorithms all make use of the local path-planning algorithms when planning and optimising paths. The path-tracking algorithm was developed during the final stages, once optimised path-planning algorithms had been developed. Its purpose was simply to demonstrate path execution. The different algorithms were developed in MATLAB and C++, with MATLAB used for fast prototyping and testing, while the C++ implementations were used to deploy the system through the Robot Operating System (ROS).

A summary of the work done towards the realisation of the of the goals of the project follows.

Chapter 1 (Introduction): This opening chapter introduces the autonomous navigation problem, highlighting the different submodules that constitute a typical autonomous navigation system and the interaction between them. Central to this discussion is the autonomous navigation (AutoNav) framework – a generalised framework used in our lab for the development of autonomous-navigation algorithms. Throughout the discussion, emphasis is placed on path planning, path optimisation and path tracking – other submodules of the framework are only discussed in enough detail to make the context and their interaction with the path-planning, path-optimisation and path-tracking submodules clear. With the autonomous navigation framework discussed and the role of path planning, path optimisation, and path tracking clarified, the chapter ends by presenting the solution approach. It does so by introducing the idea of combining path planning and

path optimisation for accelerated convergence towards the optimal solution. It also presents the proposed architecture of the optimised path-planning and path-tracking system that the project aims to develop. Project objectives and assumptions made are then derived from the proposed architecture.

Part I (Literature review and system overview) This part presents conceptual details on each category of algorithms that the project aims to develop. It also presents detailed review and evaluation of existing techniques, with the aim of selecting suitable ones that have eventually been adapted and extended in the thesis. In Chapter 2, concepts and existing literature on path-planning algorithms are presented, evaluated and suitable benchmark algorithms selected for adaptation and extension. Sampling-based path-planning algorithms are found to be the most efficient among existing techniques and, unlike the other approaches, also have the ability to generate paths that satisfy motion constraints. Furthermore, among sampling-based path-planning algorithms, single-query sampling-based path-planning algorithms were selected. Finally, among single-query sampling-based path planners, feasible, anytime, and optimal path planners were found to be of interest to our objective of optimised path planning. In path-planning terminology, feasible and anytime path planners are almost-surely suboptimal. However, their advantage over optimal path planners is their execution speed – they are quick and efficient. On the other hand existing sampling-based optimal path planners are said to be asymptotically optimal – this means that their guarantee to find the optimal path hinges on the number of iterations tending to infinity; as a result they converge slowly. This diverse set of path-planning algorithms were found to be interesting benchmarks due to two reasons. Firstly, for the almost-surely suboptimal ones there is hope for accelerated convergence towards the optimal solution if a path optimisation step is incorporated, more so because they are fast and thus allow enough time for optimisation. Secondly, for the asymptotically optimal ones, acceleration of convergence is required to make them converge in finite time. In addition to just developing path planners with inspiration from literature, the objective of the thesis then became that of investigating the effectiveness of different path optimisation algorithms in accelerating the convergence of the selected benchmark path planners. Of particular interest was to find out if the application of path optimisation to accelerate the rate of convergence can help an almost-surely suboptimal path planner attain comparable or better performance than an asymptotically optimal path planner. The different path optimisation algorithms adapted for this purpose were selected through the literature review on path optimisation presented in Section 2.8. Lastly, Section 2.7 reviewed techniques for local path planning – a component of sampling-based path planning.

With the literature for optimised path planning reviewed and suitable algorithms selected for adaptation and extension, Chapter 3 did the same for path-tracking literature.

Part II (System implementation) This is the part of the thesis where the development of algorithms solving each of the sub-problems addressed by the thesis takes place. Chapter 4 addresses the problem of computing continuous-curvature paths between two configurations in the absence of obstacles (local paths). The developed algorithm is used as a primitive by the benchmark path planners, path optimisers, as well as the optimised path planners, all developed in Chapter 5.

Benchmark path planners are the first developed in Chapter 5. Their development involves adapting them so that they produce paths that are curvature-continuous. This is achieved by applying the continuous-curvature local path planner developed in Chapter 4. The developed benchmark path planners are the basic RRT, the anytime RRT and the informed RRT*. These developed continuous-curvature (CC) versions of the benchmark path planners are respectively referred to as the CC basic RRT, CC anytime RRT and CC informed RRT*. The CC basic RRT is solely developed as foundation for the development of the other two. The other two are then the ones on which the effect of a path-optimisation step in accelerating convergence is to be investigated. From the CC anytime RRT, two extremes of the algorithm are each used as a benchmark, in addition to the CC informed RRT*, for our investigation. In one extreme, the CC anytime RRT is very quick, though almost-surely suboptimal; in the other setting, it produces better paths (though still almost-surely suboptimal), at the expense of increased computation

time. The first extreme is referred to as the CC basic anytime RRT and the second is referred to as the CC k-nearest anytime RRT. Relative to the other two benchmark path planners, the CC informed RRT* is superior in terms of the optimality of solutions. More precisely, while the other two are almost-surely suboptimal, the CC informed RRT* is asymptotically optimal. The three benchmark algorithms strive towards finding better paths using the same principle – they iteratively bound the sampled region with each newly-found solution. The idea of accelerating their convergence towards the optimal solution then works by applying path optimisation to optimise each newly-found solution before being used to bound the search space.

With the benchmark path planners developed, they are then analysed and compared. The purpose of this comparison is to establish a baseline for their head-to-head performance. This baseline is in turn used to evaluate the effectiveness of different path-optimisation algorithms in accelerating each benchmark path planners' rate of convergence. Of interest in this investigation is to ascertain if the application of path optimisation to accelerate the rate of convergence of the path planners helps a quick and efficient, though almost-surely suboptimal, path planner attain comparable or better performance than that of an asymptotically optimal path planner. Four path-optimisation algorithms have been adapted for continuous-curvature path optimisation. These are the path-pruning method, random shortcut, the wrapping process and gradient-based path optimisation. With the path-optimisation algorithms developed, they are then incorporated into the benchmark to accelerate their convergence. The result is a dozen of optimised path planners, of which there are four sets, with each set resulting from applying a given path-optimisation algorithm to accelerate each of the benchmark path planners. Finally, analyses are performed for each set of optimised path planners, comparing each of them to its baseline, and, furthermore, comparing them among each other. Interesting results have been obtained, which indicate that the application of a path-optimisation step to the selected benchmark path planners does help the almost-surely suboptimal path planner, the CC basic anytime RRT, attain comparable or better performance than the asymptotic, almost-surely optimal path planner, the CC informed RRT*.

Chapter 6 concludes the development of algorithms by developing a path-tracking algorithm that uses feed-forward and feedback control to cause a robot to follow planned continuous-curvature paths. Thereafter, Chapter 7 is devoted to the integration of the optimised path-planning algorithms with the path tracker for the purpose of demonstrating autonomous navigation capability. The C++ implementations of our algorithms have been used to deploy the integrated system through the Robot Operating System (ROS). With the integrated system in place, autonomous navigation experiments are performed to test the system's ability to plan optimised paths and accurately track them. Results show that the system is able to plan optimised paths and that the path tracking controller performs as designed – in the absence of disturbances, it causes the robot to accurately track the planned path, and it is able to correct for deviation when disturbances are encountered.

With optimised path planning and path tracking demonstrated, the goals of the project have been achieved.

8.1.2 Contributions

A summary of contributions made by this thesis is presented below. They include:

1. A consolidated review of existing techniques for optimised path planning and path tracking.
2. Adaptation of Dubins local path planner for continuous-curvature local path planning.
3. Adaptation of the basic RRT, the anytime RRT and the informed RRT* path-planning algorithms for continuous-curvature path planning.
4. Analyses and comparison of the performance of the adapted anytime RRT and the informed RRT* path planners in the third point (above).
5. Adaptation of path-optimisation algorithms including, the path pruning method, the random shortcut method, the wrapping process and a gradient-based path optimisation algorithm, for continuous-curvature path optimisation.

6. Application of the adapted path-optimisation algorithms in the fourth point (above) for accelerating the rate of convergence of the adapted anytime RRT and the informed RRT* path planners.
7. Analyses and comparison of the accelerated versions of the anytime RRT and the informed RRT* in the fifth point (above) to study the effectiveness of the different path- optimisation algorithms in accelerating their rate of convergence.
8. A path tracking controller using both feed-forward and feedback control to accurately track planned continuous-curvature paths.
9. Adaptation of the ROS navigation stack to the ESL autonomous navigation framework.

8.2 Future work

In closing, we present some future work in relation to the work presented in this thesis.

1. In this thesis, the investigation of the application of path-optimisation algorithms to accelerate the rate of convergence of path planners has been performed with the cost of a path given by its length. This idea can be extended to different cost functions in future projects.
2. While this project considered shortcut-based path optimisation and gradient-based path optimisation independently, it might be interesting to consider combining these two paradigms. If we consider the case of the path-pruning algorithm, which is quick to optimise paths, but highly depends on the visibility among consecutive nodes, it becomes interesting to consider the use of an optimiser, such as gradient descent, to optimise the placement of nodes on the node path, prior to the application of shortcuts. Other possibilities exist, but this is meant as an example.

Appendices

Appendix A

Analyses of Optimised Path Planners

A.1 Confidence Intervals

In the optimised path planning results presented in Section 5.6, confidence intervals have been used to report the statistics of the developed path planners along with bounds on the likely values of these statistics in repeated experiments. This section gives details on the concept of confidence intervals and their application to our analyses.

A confidence interval, denoted by CI, is a statistical tool used to estimate a population parameter by using statistics obtained from a sample subset of the population. It gives the range of values that, based on the sampled data, are likely to contain the true value of the population parameter [181]. The probability that the interval does contain the true value of the parameter is known as the confidence level, denoted by CL.

For a given sample statistic, ι , the confidence interval, CI, is given by the sample statistic plus or minus a *margin of error*:

$$\text{CI} = \iota \pm z^* \sigma, \quad (\text{A.1})$$

where z^* is an appropriate value (known as the z -value) from the Z -distribution, chosen according to the desired confidence level, and σ is the *standard error*. The standard error is similar to the standard deviation of a population, with the difference being that the standard error measures the variation among all possible values of the statistic under consideration while the standard deviation of a population measures the variation among all possible values that individuals can take within the population. The term $z^* \sigma$ is the margin of error. This is the term that accounts for the fact that only a subset of the population is represented by the sample statistic and thus this statistic can be expected to be off by a certain amount from the true value of the population parameter. It gives the range within which our results are accurate. Table A.1 gives a list of commonly used z -values, along with their corresponding confidence levels. As an example, if a z -value of 1.96 is used to estimate a population parameter, then it can be said that the true value of the population parameter lies within the interval $\bar{\iota} + 1.96\sigma$ with a probability of 95%, where ι and σ are again the sample statistic and the standard error.

CL	z^*
80	1.28
90	1.64
95	1.96
98	2.33
99	2.58

Table A.1: z -values for selected confidence levels.

For a population mean, the confidence interval is given by:

$$\mu = \bar{x} \pm z^* \frac{s}{\sqrt{n}}, \quad (\text{A.2})$$

where \bar{x} is the sample mean, s is the population standard deviation, n is the sample size, and z^* is as defined before.

In the context of the analyses of our optimised path planners, the confidence interval for a population mean is used to estimate, with confidence bounds, the mean path cost that each optimised path planner finds after given numbers of iterations.

Bibliography

- [1] Wired Brand Lab, “A Brief History of Autonomous Vehicle Technology,” <https://www.wired.com/brandlab/2016/03/a-brief-history-of-autonomous-vehicle-technology/>, undated, [Online; accessed 2016-03-02].
- [2] M. Weber, “Where to? A History of Autonomous Vehicles,” <http://www.computerhistory.org/atchm/where-to-a-history-of-autonomous-vehicles/>, 2014, [Online; accessed 2016-03-02].
- [3] B. Bontrage, “The Race to Fully Autonomous Cars,” <https://medium.com/swlh/the-race-to-fully-autonomous-cars-8212ff73aad>, 2018, [Online, accessed 09-11-2018].
- [4] T. Litman, “Autonomous Vehicle Implementation Predictions: Implications for Transport Planning,” *Victoria Transport Policy Institute* Tech. Report, November, 2018, [Available at: <http://www.vtpi.org/avip.pdf>, accessed 2018-12-08].
- [5] SyncedReview, “Global Survey of Autonomous Vehicle Regulations,” <https://medium.com/syncedreview/global-survey-of-autonomous-vehicle-regulations-6b8608f205f9>, 2018, [Online, accessed 09-11-2018].
- [6] Defence Advanced Projects Agency, “The DARPA Grand Challenge: Ten Years Later,” <https://www.darpa.mil/news-events/2014-03-13>, 2014, [Online; accessed 2016-05-23].
- [7] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, A. Ng, “ROS: an open-source Robot Operating System,” *ICRA Workshop on Open Source Software*, 2009.
- [8] N. Koenig, A. Howard, “Design and Use Paradigms for Gazebo, An Open-Source Multi-robot Simulator,” *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, Sendai, 2004, vol. 3, pp. 2149–2154.
- [9] L. Fridman, D. E. Brown, M. Glazer, W. Angell, S. Dodd, B. Jenik, J. Terwilliger, J. Kindelsberger, L. Ding, S. Seaman, H. Abraham, A. Mehler, A. Sipperley, A. Pettinato, B. Seppelt, L. Angell, B. Mehler, B. Reimer, “MIT Autonomous Vehicle Technology Study: Large-Scale Deep Learning Based Analysis of Driver Behavior and Interaction with Automation,” arXiv:1711.06976v2 [cs.CY], Sep. 2018.
- [10] J. Faigl, “Artificial Intelligence in Robotics: Robotic Paradigms and Control Architectures,” Lecture Notes, Czech Technical University in Prague, 2018.
- [11] T. Fraichard, A. Scheuer, “From Reeds and Shepp’s to continuous-curvature paths,” *IEEE Transactions on Robotics*, vol. 20, no. 6, pp. 1025–1035, 2004.
- [12] D. K. Wilde, “Computing clothoid segments for trajectory generation,” *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, St. Louis, MO, 2009, pp. 2440–2445.
- [13] S. Zilberstein, S. Russell, “Approximate reasoning using anytime algorithms,” *Imprecise and Approximate Computation*, Kluwer Academic Publishers, 1995.

- [14] T. Dean, M. Boddy, "An analysis of time-dependent planning," *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1988.
- [15] M. Linkhachev, G. Gordon, S. Thrun, "ARA*: Anytime A* with provable bounds on sub-optimality," *Advances in Neural Information Processing Systems*, 2003.
- [16] M. Linkhachev, M. Fegurson, G. Gordon, S. Thrun, "Anytime Dynamic A*: an anytime, replanning algorithm," *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2005, pp. 262–271.
- [17] M. Linkhachev, M. Fegurson, G. Gordon, S. Thrun, "Anytime Dynamic A*: The Proofs," Tech. Report, School of Computer Science, Carnegie Mellon University, Tech. Report, CMU-RI-TR-05-12, 2005.
- [18] D. Fegurson and A. Stentz, "Anytime RRTs," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Beijing, 2006, pp. 5369–5375.
- [19] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli and S. Teller, "Anytime Motion Planning using the RRT*," *IEEE International Conference on Robotics and Automation*, Shanghai, 2011, pp. 1478–1483.
- [20] J. D. Gammell, S. S. Srinivasa and T. D. Barfoot, "Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Chicago, IL, 2014, pp. 2997–3004.
- [21] J.M. Snider, "Automatic Steering Methods for Autonomous Automobile Path Tracking," Robotics Institute, Carnegie Mellon University, Tech. Report, CMU-RI-TR-09-08, 2009.
- [22] C. E. van Daalen, "Conflict Detection and Resolution for Autonomous Vehicles," Ph.D. Thesis, Stellenbosch University, 2010.
- [23] D. Filliat, J.A. Meyer, "Map-based navigation in mobile robots - I. A review of localisation strategies," *Journal of Cognitive Systems Research*, vol. 4, pp. 243–282, 2004.
- [24] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part I," *IEEE Robotics and Automation Magazine*, vol. 13, no. 2, pp. 99–110, 2006.
- [25] S. M. LaValle, "Planning Algorithms," *Cambridge University Press*, New York, NY, USA, 2006.
- [26] J. C. Latombe, "Robot Motion Planning," *Kluwer Academic Publishers*, 1991.
- [27] J.-P. Laumond, "Robot Motion Planning and Control," *Springer-Verlag*, Berlin, Heidelberg, 1998.
- [28] H. M. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun "Principles of Robot Motion: Theory, Algorithms, and Implementation," *MIT Press*, New York, NY, USA, 2005.
- [29] G. F. Franklin, J. D. Powell, A. Amami-Naeini, "Feedback Control of Dynamic Systems. 4th ed.," *Prentice Hall PTR*, Upper Saddle River, NJ, USA, 2001.
- [30] R. T. Stefani, B. Shahian, C. J. Savant, "Design of Feedback Control Systems. 4th ed.," *Oxford University Press, Inc.*, New York, NY, USA, 2001.
- [31] A. De Luca, G. Oriolo, and C. Samson, "Feedback control of a non-holonomic car-like robot," in *Robot Motion Planning and Control*, J.-P. Laumond, Ed. London: Springer-Verlag, 1998, pp. 171–253.
- [32] A. Gasparetto, P. Boscariol, A. Lanzutti and R. Vidoni, "Trajectory planning in robotics," *Mathematics in Computer Science*, vol. 6, no. 3, pp. 269–279, 2012.

- [33] MobileRobots Inc., "Pioneer3AT Rev. A datasheet," <https://www.generationrobots.com/media/Pioneer3AT-P3AT-RevA-datasheet.pdf>, 2011, [Online; accessed 2017-07-02].
- [34] MobileRobots Inc., "Pioneer 3 Operations Manual with MobileRobots Exclusive Advanced Robot Control & Operations Software," 2006.
- [35] J. J. Kuffner, S. M. Lavalle, "RRT-Connect: An efficient approach to single-query path planning," *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 995–1001, 2000.
- [36] S. Karaman, E. Frazzoli, "Sampling-based Algorithms for Optimal Motion Planning," *International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.
- [37] R. Guernane, N. Achour, "Generating optimized paths for motion planning," *Robotics and Autonomous Systems*, vol. 59, pp. 789–800, 2011.
- [38] Y. Guo, L. E. Parker, "A distributed and optimal motion planning approach for multiple mobile robots," *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, Washington, DC, USA, vol.3 pp. 2612-2619, 2002.
- [39] T. Lozano-Perez, "Spatial Planning: A Configuration Space Approach," *IEEE Transactions on Computers*, vol. C-32, no. 2, pp. 108–120, 1983.
- [40] A. C. Nearchou, "Path planning of a mobile robot using genetic heuristics," *Robotica*, vol. 16, pp. 575–588, 1998.
- [41] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematlk*, vol. 1, pp. 269–271, 1959.
- [42] B. Chazelle, "Approximation and decomposition of shapes," *Algorithmic and Geometric Aspects of Robotics*, pp. 145–185, Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.
- [43] S. M. LaValle, M. S. Branicky, S. R. Lindemann, "On the Relationship Between Classical Grid Search and Probabilistic Roadmaps," *The International Journal of Robotics Research*, vol. 23, no. 7, pp. 673–692, 2004.
- [44] K. Kondo, "Motion planning with six degrees of freedom by multi-strategic bidirectional heuristic free-space enumeration," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 3, pp. 267–277, 1991.
- [45] P. Yap, "Grid-Based Path-Finding," *Proceedings of the Canadian Conference on Artificial Intelligence*, pp. 44–55, 2002.
- [46] J. Lengyel, M. Reichert, and B. R. Donald, D. P. Greenberg, "Real-time Robot Motion Planning Using Rasterizing Computer Graphics Hardware," *SIGGRAPH Comput. Graph.*, vol. 24, no. 4, pp. 327–335, 1990.
- [47] A. Nash, K. Daniel, S. Koenig, A. Felner "Theta*: Any-angle path planning on grids," *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 1177–1183, 2007.
- [48] R. E. Bellman, "Dynamic Programming," *Dover Publications, Inc.*, New York , USA, ISBN: 0486428095, 2003.
- [49] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.
- [50] I. R. Nourbakhsh, M. R. Genesereth, "Assumptive planning and execution: A simple, working robot architecture," *Autonomous Robots*, vol. 3, no. 1, pp. 49–67, 1996.
- [51] A. V. Goldberg, "Point-to-Point Shortest Path Algorithms with Preprocessing," *Microsoft Research, Silicon Valley*, Tech. Report.

- [52] D. Ferguson, M. Likhachev, A. Stentz, "A guide to heuristic-based path planning," *Proceedings of ICAPS Workshop on Planning under Uncertainty for Autonomous Systems*, 2005.
- [53] A. T. Stentz, "Optimal and efficient path planning for unknown and dynamic environments," Tech. Report CMU-RI-TR-93-20, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.
- [54] A. Stentz, "Optimal and Efficient Path Planning for Partially-Known Environments," *1994 IEEE International Conference on Robotics and Automation*, pp. 3310–3317, 1994.
- [55] A. Stentz, "The D* algorithm for real-time planning of optimal traverses," Tech. Report CMU-RI-TR-94-37, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1994.
- [56] A. Stentz, "The focussed D* algorithm for real-time replanning," *Proceedings of the International Joint Conference on Artificial Intelligence*, vol. 2, pp. 1652–1659, 1995.
- [57] S. Koenig, M. Likhachev, "D* Lite," *Proceedings of the AAAI Conference of Artificial Intelligence (AAAI)*, pp. 476–483, Menlo Park, CA, USA, 2002.
- [58] V. J. Lumelsky, A. A. Stepanov, "Dynamic path planning for a mobile automaton with limited information on the environment," *IEEE Transactions on Automatic Control*, vol. 31, no. 11, pp. 1058–1063, 1986.
- [59] Y. Goto, A. T. Stentz, "Mobile robot navigation: The CMU system," *IEEE Expert*, vol. 2, no. 1, pp. 44–55, 1987.
- [60] D. Ferguson, A. Stentz, "The Delayed D* algorithm for efficient path replanning," *Proceedings of the IEEE International Conference on Robotics and Automation*, Barcelona, Spain, April 2005.
- [61] D. Ferguson, A. T. Stentz, "The Field D* algorithm for improved path planning and replanning in uniform and non-uniform cost environments," Tech. Report CMU-RI-TR-05-19, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 2005.
- [62] S. Koenig, M. Likhachev, "Improved fast replanning for robot navigation in unknown terrain," *Proceedings. IEEE International Conference on Robotics and Automation*, vol. 1, pp. 968–975, Washington, DC, 2002.
- [63] G. A. Mills-Tettey, A. T. Stentz, M. B. Dias, "DD* Lite: Efficient incremental search with state dominance," Tech. Report CMU-RI-TR-07-12, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 2007.
- [64] R. Zhou, E. A. Hansen, "Multiple sequence alignment using A*," *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2003. Student abstract.
- [65] T. Ikeda, H. Imani, "Enhanced A* algorithms for multiple sequence alignments: optimal alignments for several sequences and k-opt approximate alignments for laege cases," *Theoretical computer Science*, vol. 210 pp. 341–374, 1999.
- [66] Y. Du, D. Hsu, H. Kurniawati, W. S. Lee, S. C. Ong, S. W. Png, "A POMDP approach to robot motion planning under uncertainty," *Proceedings of the International Conference on Automated Planning & Scheduling, Workshop on Solving Real-World POMDP Problems*, Toronto, 2010.
- [67] R. Smallwood, E. Sondik, "The optimal control of partially observable Markov processes over a finite horizon," *Operations Research*, vol. 21, pp. 1071–1088, 1973.
- [68] L. P. Kaelbling, M. L. Littman, A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial Intelligence 101*, pp. 99–134, 1998.
- [69] J. Pineau, G. Gordon, S. Thrun, "Point-based value iteration: An anytime algorithm for POMDPs," *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 477–484, 2003.

- [70] J. Pineau, G. Gordon, S. Thrun, "Point-based approximations for fast POMDP solving," *Proceedings of the International Symposium on Robotics Research*, 2005.
- [71] R. S. Sutton, A. G. Barto, "Reinforcement Learning: An Introduction," *The MIT Press*, London, England, 2017.
- [72] K. P. Murphy, "Machine Learning: A Probabilistic Perspective," *The MIT Press*, Cambridge, Massachusetts London, England, 2012.
- [73] C. Papadimitriou, J. Tsitsiklis, "The complexity of Markov decision processes," *Mathematics of operations research*, vol. 12, no. 3, pp. 441–450, 1987.
- [74] O. Madani, S. Hanks, A. Condon, "On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems," *Proceedings of the 16th National Conference on Artificial Intelligence*, 1999.
- [75] M. L. Littman, J. Goldsmith, M. Mundhenk, "The computational complexity of probabilistic planning," *Journal of Artificial Intelligence Research*, vol. 9, pp. 1–36, 1998.
- [76] E. Hansen, "Solving POMDPs by searching in policy space," *Proceedings of the 14th International Conference on Uncertainty in Artificial Intelligence*, Madison, Wisconsin, USA, 1998.
- [77] N. Meuleau, K. Kim, L. Kaelbling, A. Cassandra, "Solving POMDPs by searching the space of finite policies," *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence*, Stockholm, Sweden, 1999.
- [78] N. Vlassis, M. L. Littman, D. Barber, "On the Computational Complexity of Stochastic Controller Optimization in POMDPs," 2012.
- [79] T. Smith, R. Simmons, "Point-based POMDP algorithms: Improved analysis and implementation," *Uncertainty in Artificial Intelligence*, 2005.
- [80] H. Kurniawati, D. Hsu, W. S. Lee, "SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces," *Robotics: Science and Systems*, 2008.
- [81] D. K. Grady, M. Moll, L. E. Kavraki, "Combining a POMDP Abstraction with Replanning to Solve Complex, Position-Dependent Sensing Tasks," *Proceedings of the AAAI Fall Symposium*, 2013.
- [82] V. J. Lumelsky, A. A. Stepanov, "Path-planning Strategies for a Point Mobile Automaton Moving Amidst Unknown Obstacles of Arbitrary Shape," *Algorithmica*, vol. 2, pp. 403–430, 1987.
- [83] J. Barraquand, J.-C. Latombe, "Robot Motion Planning: A Distributed Representation Approach," *International Journal of Robotics Research*, vol. 10, no. 6, pp. 628–649, 1991.
- [84] J. M. Ahuactzin, P. Bessière, E. Mazer, "The Ariadne's Clew Algorithm," *Journal of Artificial Intelligence Research*, vol. 9, pp. 295–316, 1998.
- [85] D. Hsu, J.-C. Latombe, R. Motwani, "Path planning in expansive configuration spaces," *International Journal Computational Geometry & Applications*, vol. 4 pp. 495–512, 1999.
- [86] S. Carpin, G. Pillonetto, "Robot motion planning using adaptive random walks," *IEEE Transactions on Robotics & Automation*, vol. 21, no. 1, pp. 129–136, 2005.
- [87] S. Carpin, G. Pillonetto, "Merging the adaptive random walks planner with the randomized potential field planner," *IEEE International Workshop on Robot Motion and Control*, pp. 151–156, 2005.
- [88] S. M. LaValle, "Motion Planning: The Essentials," *IEEE Robotics and Automation Magazine*, vol. 18, no. 1, pp. 79–89, 2011.

- [89] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Tech. Report TR 98-11, Computer Science Department, Iowa State University, 1998.
- [90] S. M. LaValle, J. J. Kuffner, "Randomized Kinodynamic Planning," *IEEE International Conference on Robotics and Automation*, pp. 473–479, 1999.
- [91] S. M. Lavalle and J. J. Kuffner, "Rapidly-Exploring Random Trees: Progress and Prospects," *Algorithmic and Computational Robotics: New Directions*, pp. 293–308, 2000.
- [92] S.M. LaValle, "From Dynamic Programming to RRTs : Algorithmic Design of Feasible Trajectories," *Control Problems in Robotics*, Springer Berlin Heidelberg, pp. 19–37, 2003.
- [93] F. Islam, J. Nasir, U. Malik, Y. Ayaz, O. Hasan, "RRT*-Smart: Rapid convergence implementation of RRT* towards optimal solution," in *IEEE International Conference on Mechatronics and Automation*, Chengdu, China, pp. 1651–1656, 2012.
- [94] J. Nasir, F. Islam, Y. Ayaz, "Adaptive Rapidly-exploring Random Tree Star (RRT*)-SMART: Algorithm Characteristics and Behaviour Analysis in Complex Environments," *Asia-Pacific Journal of Information Technology and Multimedia*, vol. 2, no. 2, pp. 39–51, 2013.
- [95] N. A. Wedge, M. S. Branicky, *On Heavy-tailed Runtimes and Restarts in Rapidly-exploring Random Trees*, 2008.
- [96] S. R. Lindemann, S. M. LaValle, "Current Issues in Sampling-based Motion Planning," *In Proceedings of the 8th International Symposium on Robotics Research*, Berlin, Germany: Springer-Verlag, pp .36-54, 2004.
- [97] C. Urmson, R. Simmons, "Approaches for heuristically biasing RRT growth," *Proceedings of the IEEE/RSJ International Conference on Robotics and Systems (IROS)*, 2003.
- [98] M.-C. Kim and J.-B. Song, "Informed RRT*:Towards Optimality by Reducing Size of Hyper-ellipsoid," *Proceedings of the IEEE International Conference on Advanced Intelligent Mechatronics (AIM)*, Busan, Korea, 2015.
- [99] S. Karaman, E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning," *International Journal of Robotics Research*, 2010.
- [100] S. Karaman, E. Frazzoli, "Sampling-based motion planning with deterministic μ calculus specifications," *IEEE Conference on Decision and Control (CDC)*, 2009.
- [101] B. Akgun, M. Stilman, "Sampling heuristics for optimal motion planning in high dimensions," *Proceedings of the IEEE/RSJ International Conference on Robotics and Systems (IROS)*, pp. 2640–2645, 2011.
- [102] M. Otte, N. Correll, "C-FOREST: Parallel shortest path planning with superlinear speedup," *TRO*, vol. 29, no. 3, pp. 798–806, 2013.
- [103] J. Bruce, M. Veloso, "Real-time randomized path planning for robot navigation," *IEEE Conference of Automation Science and Engineering*, 2002.
- [104] D. Ferguson, N. Kalra, A. Stentz, "Replanning with RRTs," *IEEE International Conference on Robotics and Automation (ICRA)*, 2006.
- [105] M. Zucker, J. Kuffner, M. Branicky, "Multipartite RRTs for rapid replanning in dynamic environments," *IEEE International Conference on Robotics and Automation (ICRA)*, 2007.
- [106] M. Otte, E. Frazzoli, "RRT^X: Asymptotically Optimal Single-Query Sampling-Based Motion Planning with Quick Replanning," *Massachusetts Institute of Technology*, Cambridge MA 02139, USA, 2010.
- [107] L. E. Dubins, "On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents," *American Journal of Mathematics*, vol. 79, no. 3, pp. 497–516, 1957.

- [108] J. Reeds, L. Shepp, "Optimal paths for a car that goes both forwards and backwards," *Pacific Journal of Mathematics*, vol. 145, no. 2, pp. 367–393, 1990.
- [109] X.-N. B. Xuan, J. D. Boissonnat, P. Soueres, J.-P. Laumond, "Shortest path synthesis for Dubins nonholonomic robot," *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, vol. 1, pp. 2–7, 1994.
- [110] A. Scheuer, T. Fraichard, "Continuous-curvature path planning for car-like vehicles," *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS '97*, vol. 2, pp. 997–1003, 1997.
- [111] A. Scheuer, T. Fraichard, "Collision-Free and Continuous-Curvature Path Planning for Car-Like Robots," 1997.
- [112] T. Fraichard, J. M. Ahuactzin, "Smooth path planning for cars," *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 4, pp. 3722–3727, 2001.
- [113] G. S. Cho, J. K. Ryeu, "An efficient method to find a shortest path for a car-like robot," *International Journal of Multimedia and Ubiquitous Engineering*, vol. 1, no. 1, pp. 1–6, 2006.
- [114] X. Z. Gao, Z. X. Hou, X.F. Zhu, F. Xiong, J. T. Zhang, X. Q. Chen, "The shortest path planning for manoeuvres of UAV," *Acta Polytechnica Hungarica*, vol. 10, no. 1, pp. 221–239, 2013.
- [115] R. G. Sanfelice, E. Frazzoli, "On the optimality of dubins paths across heterogeneous terrain," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4981 LNCS, no. 1, pp. 457–470, 2008.
- [116] C. Yong, E. J. Barth, "Real-time Dynamic Path Planning for Dubins' Nonholonomic Robot," *Proceedings of the 45th IEEE Conference on Decision and Control*, 2006.
- [117] V. Diot, "Path Tracking Control for Dubin's Cars," *Proceedings of the 1996 IEEE International Conference on Robotics and Automation*, pp. 23–28, 1996.
- [118] S. Hota, and D. Ghose, "A modified dubins method for optimal path planning of a miniature air vehicle converging to a straight line path," *Proceedings of the American Control Conference*, pp. 2397–2402, 2009.
- [119] W. Visser, "Automation and Navigation of a Terrestrial Vehicle," Masters Thesis, Stellenbosch University, 2012.
- [120] H. J. Sussmann, G. Tang, "Shortest Paths For The Reeds-Shepp Car: A Worked Out Example Of The Use Of Geometric Techniques In Nonlinear Optimal Control.," 1991.
- [121] H. J. Sussmann, "The Markov-Dubins problem with angular acceleration control," *Proceedings of the IEEE Conference on Decision and Control*, pp. 2639–2643, 1997.
- [122] J.-D. Boissonnat, A. Cérézo, J. Leblond, "A Note on Shortest Paths in the Plane Subject to a Constraint on the Derivative of the Curvature," *Inria*, 1994.
- [123] V. P. Kostov, E. V. Degtiariova-Kostova, "The planar motion with bounded derivative of the curvature and its suboptimal paths," *Acta Math. Univ. Comenianae*, vol. LXIV, pp. 185–226, 1995.
- [124] E. Bakolas, P. Tsiotras, "On the generation of nearly optimal, planar paths of bounded curvature and bounded curvature gradient," *Proceedings of the American Control Conference*, pp. 385–390, 2009.
- [125] M. Brezak, I. Petrovic, "Real-time approximation of clothoids with bounded error for path planning applications," *IEEE Transactions on Robotics*, vol. 30, no. 2, pp. 507–515, 2014.

- [126] V. A. Girbes, L. Armesto, J. Tornero, "On generating continuous-curvature paths for line following problem with curvature and sharpness constraints," *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 6156–6161, 2001.
- [127] K. R. Meidenbauer, "An Investigation of the Clothoid Steering Model for Autonomous Vehicles An Investigation of the Clothoid Steering Model for Autonomous Vehicles," 2007.
- [128] "Planning continuous curvature paths using constructive polylines," *Journal of Aerospace Computing, Information and Communication*, vol. 4, no. 12, pp. 1143–1157, 2007.
- [129] D. S. Meek, D. J. Walton, "A note on finding clothoids," vol. 171, pp. 433–453, 2004.
- [130] K. Yang, D. Jung, S. Sukkariéh, "Continuous curvature path-smoothing algorithm using cubic Bezier spiral curves for non-holonomic robots," 2013.
- [131] A. M. Lekkas, A. R. Dahl, M. Breivik, T. I. Fossen, "Continuous-Curvature Path Generation Using Fermat's Spiral," 2013.
- [132] J. Nocedal, S. J. Wright, "Numerical Optimization," *Springer Series in Operations Research*, Ed. P. Glynn and S. M. Robinson, Springer-Verlag New York, 1999.
- [133] R. Geraerts, M. H. Overmars, "Creating High-quality Paths for Motion Planning," *International Journal of Robotics Research*, vol. 26, no. 8, pp. 845–863, 2007.
- [134] S. Sekhavat, P. Svestka, J. -P. Laumond, M. H. Overmars, "Multi-Level Path Planning for Nonholonomic Robots using Semi-Holonomic Subsystems," *International Journal of Robotics Research*, vol. 17, pp. 840–857, 1996.
- [135] K. Hauser, V. Ng-Thow-Hing, "Fast smoothing of manipulator trajectories using optimal bounded-acceleration shortcuts," *IEEE International Conference on Robotics and Automation (ICRA)*, Anchorage(AK), pp. 2493–2498, 2010.
- [136] M. Campana, F. Lamiriaux, J-P. Laumond, "A gradient-based path optimization method for motion planning," *Advanced Robotics*, vol. 30, no. 17–18, pp. 1126–1144, DOI: 10.1080/01691864.2016.1168317, 2016.
- [137] N. Ratliff, M. Zucker, J. A. Bagnell, S. Srinivasa, "CHOMP: Gradient Optimization Techniques for Efficient Motion Planning," *IEEE Conference on Robotics and Automation (ICRA)*, pp. 489–494, 2009.
- [138] M. Zucker, N. D. Ratliff, A. D. Dragan, M. Pivtoraiko, M. Klingensmith, C. M. Dellin, J. A. Bagnell, S. S. Srinivasa, "CHOMP: Covariant Hamiltonian Optimization for Motion Planning," *International Journal of Robotics Research*, vol. 32, no. 9–10, pp. 1164–1193, 2013.
- [139] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, S. Schaal, "STOMP: Stochastic trajectory optimization for motion planning," *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, 2011.
- [140] C. Park, J. Pan, D. Manocha, "ITOMP: incremental trajectory optimization for real-time replanning in dynamic environments," *International Conference on Automated Planning and Scheduling (ICAPS)*, Atibaia, São Paulo, Brazil, pp. 207–215. 2012.
- [141] B. Bogaerts, S. Sels, S. Vanlanduit, R. Penne, "A Gradient-Based Inspection Path Optimization Approach," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, July 2018.
- [142] Y. Shan, C. Chen, J. Zhou, L. Zheng, B. Li, W. Yang, "CF-Pursuit: A Pursuit Method with a Clothoid Fitting and a Fuzzy Controller for Autonomous Vehicles," *International Journal of Advanced Robotic Systems*, vol. 12, no. 134, pp. 1–13, 2015.
- [143] J. L. Giesbrecht, D. Mackay, J. Collier, S. Verret, "Path Tracking for Unmanned Ground Vehicle Navigation," *Defence Research and Development Canada*, Tech. Report, 2005.

- [144] B. Paden, M. Cáp, S. Z. Yong, S. Y. Dmitry, E. Frazzoli, “A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles,” *Conference on Robotics Research*, 2016.
- [145] J. S. Wit, “Vector Pursuit Path Tracking for Autonomous Ground Vehicles,” PhD Thesis, University of Florida, 2000.
- [146] L. Scharf, W. Harthill, and P. Moose, “A comparison of expected flight times for intercept and pure pursuit missiles,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 4, pp. 672–673, 1969.
- [147] R. Wallace, A. Stentz, C. Thorpe, H. Maravec, W. Whittaker, T. Kanade, “First Results in Robot Road-following,” *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, vol. 2, pp. 1089–1095, 1985.
- [148] O. Amidi, “Integrated Mobile Robot Control,” Tech. Report CMU-RI-TR-90-17, Carnegie Mellon University, 1990.
- [149] R. C. Coulter, “Implementation of the Pure Pursuit Path Tracking Algorithm,” *Carnegie Mellon University, Robotics Institute*, Tech. Report, 1992.
- [150] J. Wit, C. D. Crane, D. Carl, D. Armstrong, “Autonomous ground vehicle path tracking,” *Journal of Robotic Systems*, vol. 21, no. 8, pp. 439–449, 2004.
- [151] R. S. Ball, “A Treatise on the Theory of Screws,” Cambridge University Press, Cambridge, United Kingdom, 1900.
- [152] Y. Shan, W. Yang, C. Chen, J. Zhou, L. Zheng, and B. Li, “CF-Pursuit: A Pursuit Method with a Clothoid Fitting and a Fuzzy Controller for Autonomous Vehicles,” *International Journal of Advanced Robotic Systems*, vol. 12, 2015.
- [153] V. Gírbés, L. Armesto, J. Tornero, J. E. Solanes, “Continuous-Curvature Kinematic Control for Path Following Problems,” *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4335–4340, 2011.
- [154] V. Gírbés, L. Armesto, J. Tornero, “Path following hybrid control for vehicle stability applied to industrial forklifts,” *Robotics and Autonomous Systems*, vol.62, pp. 910–922, 2014.
- [155] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, . Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L. -E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, P. Mahoney, “Stanley: The Robot That Won the DARPA Grand Challenge: Research Articles,” *Journal of Robotic Systems*, vol. 23, no. 9, pp. 661–692, 2006.
- [156] W. van den Aardweg, “Robust Sampling-Based Conflict Resolution for Commercial Aircraft in Airport Environments,” Masters Thesis, Stellenbosch University, 2015.
- [157] M. Strandberg, “Augmenting RRT-planners with local trees,” *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 4, pp. 3258–3262, 2004.
- [158] R. Kala, “Rapidly exploring random graphs: motion planning of multiple mobile robots,” *Advanced Robotics*, vol. 27, no. 14, pp. 1113–1122, 2013.
- [159] L. Kavraki, P. Svestka, J.-C. Latombe, M. Overmars, “Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces,” Tech. Report, Stanford University, Stanford, CA, USA, 1994.
- [160] S. Karaman, E. Frazzoli, “Sampling-based algorithms for optimal motion planning,” *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.
- [161] S. Thrun, W. Burgard, Wolfram, D. Fox, “Probabilistic Robotics (Intelligent Robotics and Autonomous Agents),” *MIT Press*, 2005.

- [162] “Planning Collision-Free Reaching Motions for Interactive Object Manipulation and Grasping,” 2003.
- [163] J. Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, P. Abbeel, “Motion Planning with Sequential Convex Optimization and Convex Collision Checking,” *International Journal of Robotics Research*, vol. 33, no. 9, pp. 1251–1270, 2014.
- [164] E. M. T. Hendrix, B. G. Tóth, “Introduction to Nonlinear and Global Optimization,” *Springer Optimization and its Applications*, Ed. P. M. Pardalos and D. Z. Du, Springer New York, 2010.
- [165] M. Orsag, C. Korpela, P. Oh, S. Bogdan, “Coordinate Systems and Transformations,” *Aerial Manipulation. Advances in Industrial Control, Springer, Charm*, pp. 19–31, 2018.
- [166] D. S. Meek, D. J. Walton, “An arc spline approximation to a clothoid,” *Journal of Computational and Applied Mathematics*, vol. 170, pp. 57–77, 2004.
- [167] T. E. Marlin, “Process Control: Designing Processes and Control Systems for Dynamic Performance,” *McGraw-Hill* 1995.
- [168] MobileRobots LLC “ARIA Developer’s Reference Manual,” <http://robots.mobilerobots.com/docs/api/ARIA/2.9.1/docs/index.html> [online, accessed 07-02-2017].
- [169] Online, “ROS Introduction,” <http://wiki.ros.org/ROS/Introduction> [online; accessed 07-02-2017]
- [170] Online, “ROS Technical Overview,” <http://wiki.ros.org/ROS/Technical%20Overview> [online; accessed 07-02-2017]
- [171] Online, “Tutorial: ROS Control,” http://ros.org/wiki/ros_control [online; accessed 07-02-2017]
- [172] Online, “ROS Concepts,” <http://wiki.ros.org/ROS/Concepts> [online; accessed 07-02-2017]
- [173] Online, “Setup and Configuration of the Navigation Stack on a Robot,” http://wiki.ros.org/navigation/Tutorials/RobotSetup#Robot_Setup
- [174] L. Nogueira, “Comparative Analysis Between Gazebo and V-REP Robotic Simulators,” Tech. Report, Universidade de Campinas, 2014.
- [175] MobileRobots LLC, “MobileSim Simulator,” [urlhttp://robots.mobilerobots.com/wiki/MobileSim](http://robots.mobilerobots.com/wiki/MobileSim) [online; accessed 07-02-2017].
- [176] M. Freese, S. Singh, F. Ozaki, N. Matsuhira, “Virtual Robot Experimentation Platform V-REP: A Versatile 3D Robot Simulator,” *Proceedings of the Second International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, Darmstadt, Germany, pp. 51–62, 2010.
- [177] Marguedas, “How to use ROSARIA,” <http://wiki.ros.org/ROSARIA/Tutorials/How%20to%20use%20ROSARIA> [online; accessed 07-02-2017].
- [178] A. Staranowicz, G. L. Mariottini, “A Survey and Comparison of Commercial and Open-source Robotic Simulator Software,” *Proceedings of the 4th International Conference on Pervasive Technologies Related to Assistive Environments*, 2011.
- [179] “vrep_ros_bridge,” http://wiki.ros.org/vrep_ros_bridge, [online; accessed 11-02-2017].
- [180] A. Martinez, E. Fernandez, “Learning ROS for Robotics Programming,” *Published by Packt Publishing Ltd*, Birmingham B3 2PB, UK, 2013.
- [181] D. G. Altman, D. Machin, T. N. Bryant, M. J. Gardner, “Statistics With Confidence: Confidence intervals and statistical guidelines,” *British Medical Journal*, 2000.